Challenger: A Multi-agent System for Distributed Resource Allocation

Anthony Chavez, Alexandros Moukas and **Pattie Maes**
Autonomous Agents Group
MIT Media Lab
20 Ames Street
Cambridge, MA 02139-4307
asc/moux/pattie@media.mit.edu

Abstract

In this paper we introduce Challenger, a multi-agent system that performs completely distributed resource allocation. Challenger consists of agents which individually manage local resources; these agents communicate with one another to share their resources (in this particular instance, CPU time) in an attempt to more efficiently utilize them. By endowing the agents with relatively simple behaviors which rely on only locally available information, desirable global system objectives can be obtained, such as minimization of mean job flow time. Challenger is similar to other market-based control systems in that the agents act as buyers and sellers in a marketplace, always trying to maximize their own utility. The results of several simulations of Challenger performing CPU load balancing in a network of computers are presented. The main contribution of this research is the addition of learning to the agents, which allows Challenger to perform better under a wider range of conditions than other systems for distributed processor allocation, such as Malone's Enterprise [Mal88].

Keywords: resource allocation, multi-agent system

1 Introduction

Computer systems are becoming increasingly complex, which has led researchers to come up with new techniques for controlling this complexity. One such technique is *market-based control*. In the words of one of its proponents, "market-based control is a paradigm for controlling complex systems that would otherwise be very difficult to control, maintain, or expand" [Cle96a]. As its name suggests, market-based control works by the same principles that real economic markets do: through the interaction of local agents, coherent global behavior is achieved. The agents trade with one another using a relatively simple mechanism, yet desirable global objectives can often be realized, such as stable prices or efficient resource allocation [Cle96a]. The fundamental appeal of the market as a model for managing complex systems is its ability to yield desirable global behavior on the basis of agents acting on only locally available information.

We have designed a multi-agent system, named *Challenger*, that performs distributed resource allocation (in particular, allocation of CPU time), using market-based control techniques. In this paper, we describe the architecture of Challenger and simulations which we have conducted. Section 2 gives some background and motivation. Section 3 describes the low-level base architecture of the Challenger agents. Section 4 presents simulation results of Challenger conducted with only the agents' base behaviors activated. Section 5 describes the learning behaviors of the Challenger agents which allow the system to perform better under a wider range of operating conditions and presents results which confirm this. Section 6 discusses other systems that do distributed processor allocation, such as Tom Malone's Enterprise system [Mal88], and compares them to Challenger. Section 7 briefly talks about future work and concludes.

2 Background and Motivation

We make the following bold claim: the average workstation or PC is often underutilized. We don't have any hard proof to back this up, just some anecdotal evidence.

At the MIT Media Lab (the authors' residence), one can walk around in late evening and early morning and see a large number of apparently unused and idle workstations. Perhaps a few are doing intensive jobs, and some are web servers, but most have a load of around zero. Yet, a user might be running tasks on his or her own workstation (we have personally experienced this many times), with the load so bad it takes seconds to bring up a new window. He or she could log onto other machines and run jobs on them, but only if they have access to those machines. Also, it's a hassle.

What is needed is a seamless, transparent way of utilizing the unharnessed computing power of a networked community. When a user creates a new task, that task should be able to run locally, or, if the originating machine is experiencing high load, run on a remote machine which is currently underutilized. This should happen transparently to the user, who doesn't care if the job ran on their own workstation or on one down the hall. Challenger is designed to meet this need.

Challenger is a software agent that does distributed processor allocation. In doing processor allocation, there are different global objectives one can strive for. The three that are usually considered are: minimization of mean flow time, maximization of processor utilization, and minimization of mean response ratio [Tan92]. Mean flow time is the average time from when a job is originated to when it is completed. Processor utilization is the percentage of time a processor spends executing jobs. Mean response ratio is the average ratio of the actual time to complete a job divided by the time to run that job on an unloaded benchmark processor. In designing Challenger, our goal was to minimize mean flow time, since this seems to be what affects user satisfaction the most — people want their tasks to finish as quickly as possible!

In addition to minimizing mean flow time, Challenger was designed to have the following properties:

- *Robust*: The system should have no single point-of-failure. Traditional centralized schedulers do not meet this requirement; if the main "scheduler" goes down, the whole system fails. Thus the scheduling system needs to be decentralized.
- *Adaptive*: It is essential that the system be able to adapt quickly to changing network and machine conditions. One should not assume a static, fixed environment. The system should function minimally well even in the worst possible operating conditions.

Software agents can be broadly classified into two categories: *user* agents and *service* agents. Challenger falls into the latter. User agents assist users with specific tasks and typically interact with them in some way. A good example of a user agent is the Maxims email-prioritization agent developed at the MIT Media Lab [Mae94]. Service agents generally run in the background and assist the user, not directly, but implicitly, by making their environment a better place to work. The agent might do this by more equitably and efficiently distributing resources. An example of such a service agent is the system built by Clearwater et al. [Cle96b], which manages air conditioning within a building. Their agent distributes cooled air in a way that is fairer and more efficient (i.e. it conserves energy) than conventional systems. Challenger improves the user's environment, not by keeping them at a comfortable temperature, but by running their jobs faster.

3 Base Agent Architecture

In this section, we describe the low-level, base architecture of Challenger agents.

3.1 Operating Environment

Challenger is intended to be used in a relatively modest-sized network of workstations/PCs, somewhere on the order of 2-10 machines. While we think the system will scale to much larger networks, we suspect that a smaller network would be one on which a real Challenger could most readily be deployed in the near future. Also, as we will discuss later, there is evidence that adding more than about ten machines to a network does not improve performance, assuming that the additional machines cause more jobs to be generated.

3.2 Base Agent Behavior

Challenger is completely decentralized. It consists of a distributed set of agents, each of which runs locally on every machine in the network. There is no single point of control, or failure. Each agent is responsible for both assigning tasks originating on the local machine and allocating its processing resources. All the agents are identical in that they have the exact same behavior.

The base agent behavior is based on a market/bidding metaphor, which is summarized as follows:

- *Job origination*: When a job is originated, the local Challenger agent broadcasts a "request for bids" to all the agents in the network (including itself). This message contains a job id, a priority value, and (optionally) information that can be used to estimate how long it will take to complete the job.
- *Making bids*: If an agent is idle (i.e. if the local processor is idle) when it receives a request for bids, it responds with a bid giving the estimated time to complete the job on the local machine (calculated, if necessary, using the optional information contained in the request for bids message). If the agent is busy, i.e. running a job, when it receives the request for bids, it stores the request in a queue in order of priority. When the agent eventually becomes idle, it retrieves the highest priority request and submits a bid on it. Once an agent submits a bid on a job, it is deemed busy and waits for a response[1]
- *Evaluation of bids*: After an "evaluation delay", the originating agent evaluates all the bids it has received and assigns the task to the best bidder, i.e. the one which returned the lowest estimated completion time. Cancel messages are sent to all other agents. The evaluation delay is an adjustable parameter whose effect on overall system performance will be discussed in more detail later. If no bids have been received when the agent evaluates bids, then the job is assigned to the next agent which submits a bid.

[1] If the agent does not receive a response (either a job assignment message or a cancel message) within a reasonable timeout period, then it reverts to idle and assumes that the sender of the request for bids is inaccessible. All simulations in this paper assume that all agents stay active and do not fall off the network.

- *Returning results*: When an agent has completed a job, it returns the result to the originating agent[2].

3.3 Domain Assumptions

There are several issues regarding the above that need to be addressed:

- *Nature of jobs*: The above description of agent behavior implies that jobs are one-shot affairs; that is, they have definitely finite duration. This is well-suited for describing jobs like compilations, or vision processing runs, but it doesn't fit more open-ended tasks such as using an editor, putting up a clock, or displaying a load balance window. For instance, it would be impossible to estimate how long it would take to "run" an editor, because that depends on the user. Thus, Challenger only operates with jobs that fit the specified model. Also, in the Challenger model only one job executes on a processor at a time. This is not to say that the processor is really only executing a single job — it will still be time-sharing, running many tasks. It is just that from the viewpoint of a Challenger agent, only one Challenger-type job at a time can run on the processor-resource that it manages. But there are many jobs which run outside of the Challenger world, including the Challenger agent itself. The owner of the workstation may still be using it while Challenger runs jobs for her and others in the background.
- *Job priorities*: Jobs are assigned priorities by the local agent for the machine on which the originate. How priorities are to be assigned is determined by the overall system performance desired. Since we are trying to minimize the mean flow time (MFT), we use the heuristic of giving highest priority to jobs with the shortest estimated processing time [Mal88]. This means that the originating processor has to make an estimate of how long the job will take to complete, in order to be able to assign it a priority value.
- *Estimating job completion times*: Agents making a bid must estimate how long it would take them to complete the job should it be assigned to them. In the Challenger simulation we have built, agents make these estimates in a very simple way. Job priority values (assigned by the originating agent) are given in terms of how long it would take to complete the job on an unloaded baseline machine of speed 1. To make its estimate, the agent takes this value and divides by its own machine's speed. So, for a machine twice as fast as the baseline, the agent would estimate a completion time of half as long as that given by the job's priority value.

There is still the issue of how the originating agent makes an estimate of the job's completion time. This depends to a large extent on the nature of the job. If the job is a compilation, we might use the number of lines of code and the number of files to link as a guide. If the job is a vision processing run, we might consider the size of the image to be processed. There will undoubtedly be errors in these estimations. The simulation runs presented in the next section show the effect of such estimation errors on overall system performance.

4 Simulation Results with Base Agent Behavior

We have so far described the base agent behavior. It is quite similar to the DSP protocol of Malone's Enterprise system [Mal88]. Challenger is perfectly functional with its agents only running the base behavior; we refer to the system then as being in BASE mode. Before we present any simulation results, we briefly describe how the simulations were done.

4.1 The Simulator

We wrote a Challenger simulator[3] to test our agent architecture.

The basic way the simulations were done was to first generate a large batch of jobs, each of which was assigned a "true" length. The jobs were then fed to the simulated Challenger system, which was run until all jobs were completed.

Job arrival times were generated via a Poisson process. The inter-arrival time was adjusted so as to achieve the desired system load, or utilization. Individual job lengths (the time to run the job on an unloaded processor of speed 1) were assumed to follow an exponential distribution, with a mean time of 60 time units[4].

We used Malone's definition of system utilization: the expected amount of processing requested per time unit divided by the total amount of processing power in the system. In equation form, this is:

$$\frac{\lambda\mu}{T} = L \tag{1}$$

where λ is the average number of job arrivals per time unit, μ is the average job length (in this case, 60), T is the total processing power in the system, and

[2] If the agent does not receive the result from the agent it sent the job to within a reasonable amount of time, then it assumes that the agent has gone dead, and it should either run the job locally or send out a new request for bids for that job. Again, all simulations in this paper assume that agents stay alive and accessible.

[3] We wrote it in Java. This turned out to be a regrettable choice, since Java runs exceedingly slow. We think that Java, though, may be a good platform on which to implement a real Challenger system. See section 7.

[4] 60 was chosen as the mean to facilitate easy comparison to simulations of Malone's Enterprise, which also used this value.

L is the system load. Given this formula, it is easy to compute the λ needed to get the desired L.

We also had to assign jobs to the processors on which they "originate". We first generated a large batch of unassigned jobs given the desired system utilization. We then randomly "assigned" the jobs to machines, using a simple weighting scheme: faster machines should originate more jobs than slower ones. For example, a machine of speed 2 should originate twice as many jobs as a machine of speed 1. The rationale behind this is that a faster machine is likelier to have more people using it than a slower one. There are certainly times when this is not the case; it is not entirely clear what the best model of job generation is to use.

In the simulations, the costs of running the agents are not factored in, since they are assumed to be negligible relative to the costs of executing jobs. Additionally, all machines are assumed to be equally loaded; that is, the load due to non-Challenger jobs is the same for each machine. We explicitly state when we deviate from this assumption.

We constructed a simple graphical Java interface to our Challenger simulator that allows us to view the current state of any agent in the system while the simulation is running. Information displayed includes the current mean flow time of jobs originated on the local machine, the current length of the agent's request for bids queue, and the current average processor utilization (for Challenger jobs only) since the simulation started. Being able to see this information in a pseudo real-time fashion proved quite insightful and useful when we were trying to adjust the agent's behavior to improve system performance.

All of the simulation runs discussed below were conducted with the Challenger agents in BASE mode.

4.2 Effect of error estimation

Figure 1 shows a plot of mean flow time (MFT) versus system utilization for Challenger in configuration 41111[5]. The solid line indicates results when the agents on which the jobs originate make perfect estimates about how long they will take to run. As one can see, the higher the system utilization, i.e. the more jobs being generated per time interval, the larger the MFT. Not surprising. What is interesting is that even when the originating agents make highly inaccurate job length estimates (errors uniformly ranging from +100 to -100 percent), thus affecting job priority values and the ordering of agents' request for bid queues, performance is only barely worse than in the perfect estimate case. This is shown by the dashed line. The results in Figure 1 conform closely to those of Malone [Mal88]. For the remainder of the simulation runs in this paper, the originating agent's job length estimates are assumed to be perfect.

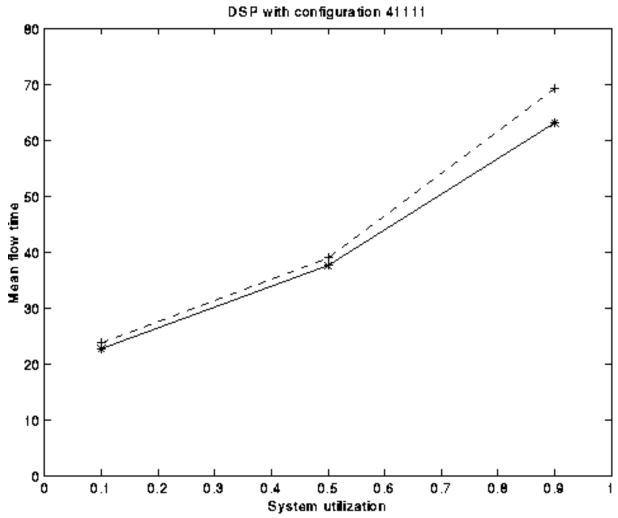

Figure 1: Configuration 41111. Results with originating agents making perfect and imperfect job length estimates.

4.3 Effect of adding processors

Figure 2 shows the effect on MFT of adding machines to a Challenger network while keeping overall system utilization constant, for three different levels of utilization: 90, 50, and 10 percent. Each machine in the network has speed 1. The MFT goes down as machines are added, as one would expect, but the curves flatten out at around 8 to 10 machines. One might ask, shouldn't continuing to add machines always improve performance? The answer is: not necessarily, because the assumption here is that the arrival of a new machine creates more jobs because of its presence. This is a rather significant result, implying that the ideal size for a network of machines *for which there will almost always be an equal number of users* is around 8 to 10. More machines beyond this doesn't improve performance significantly. These results agree with those that Malone got [Mal88]. Of course, if one adds more machines to a network without a corresponding increase in job generation, then system performance will always improve, everything else (such as network load) remaining equal.

4.4 Effect of message delay

Figures 3 and 4 show the effect on MFT of message delay, i.e. the delay between the transmission and receipt of a message caused by network lag. These figures are somewhat confusing but informative. Each figure has three sets of three lines each. Each of these sets of lines is distinguished by their type: solid, dashed, and

[5] **Explanation of notation for describing network configurations.** We pirated Malone's notation for describing these configuration concisely. For example, 41111 denotes a network consisting of 5 machines, 4 of speed 1 (the baseline machine speed), and 1 machine of speed 4. The total processing power in this setup is 8 (4+1+1+1+1).

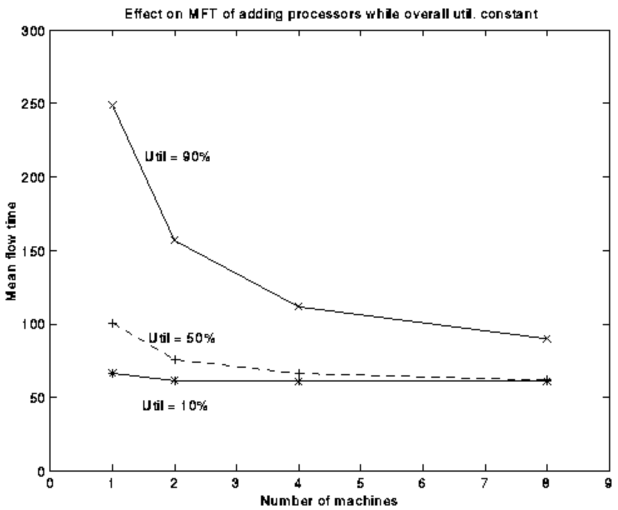

Figure 2: Effect on MFT of adding machines while keeping overall system utilization constant. Each machine has speed 1. Note that the improvement in MFT levels off around 8 machines.

dashed-dot. Each type of line represents the network in a different configuration: solid is configuration 41111, dashed is configuration 44, and dashed-dot is configuration 11111111. These configurations were chosen so as to facilitate easy comparison with Malone's Enterprise simulations, which used the same configurations [Mal88]. For each network configuration, there are three "setups" the simulations were run in:

- **Loc**: The network was turned off and each job was run on its local machine. This mode serves as a baseline for comparison to the other setups.
- **Net**: The network was on, and Challenger was run with the agents in the default BASE mode.
- **Opt**: The network was on, and Challenger was run with the agents in NTWRK mode. Note: this mode of agent operation has not been described yet, so please ignore it for now.

Each line in the figures is indicated by a label consisting of the configuration, followed by a period, followed by the setup. For now, ignore those lines that are marked Opt.

Message delay was measured not in absolute terms, but as a percent of the average true job completion time, i.e. the time it would take to run the job on an unloaded baseline machine of speed 1. We used a true job completion time of 60, so a message delay of 5 percent would be 3. Message delays are one-way (from sender to receiver) and are assumed to be fixed.

Figure 3 shows the effect of message delay on MFT for the three different configurations (44, 41111, and 11111111) with a system utilization of 10 percent. The evaluation delay (recall, the time an agent waits before evaluating bids on a job after sending a request for bids message) was set to 1. This decision will be explained shortly. For very low message delays (e.g., zero), performance with Challenger activated (in BASE mode) was always better than running all the jobs locally. As message delay increases, the performance of Challenger worsens, in all configurations becoming worse than the setup where all jobs are run locally. The exact point at which this happens depends on the configuration, but nonetheless is inevitable as message delays get large enough.

Figure 4 is very similar to Figure 3. MFT increases almost linearly as a function of message delay. Except for the 44 configuration, performance does not become worse than the setup where jobs are run locally, but this would happen if the message delay became large enough.

What do we make of these results? First, they correspond closely to the results Malone got running Enterprise under conditions of high message delays (not surprising, given the similarity between his system and the BASE mode behavior of Challenger's agents) [Mal88]. Second, they are highly undesirable. At the beginning of the paper, we said that we wanted Challenger to be *robust* and *adaptive*, and running in BASE mode, it is not. It is not robust because it does not deal with changing operating conditions well. You might say, well, just run Challenger only when you know that message delays will be small relative to average job lengths. In reality, though, it is often very difficult to guarantee that conditions will remain so static. What if the message delay goes up (say, because more machines were added to the network)? Or what if average job length goes down? You could turn off Challenger when its performance gets too poor, but then it fails to meet the requirement of a service agent of being able to run in the background without user intervention. It then becomes just a switch to flip. We would really like Challenger to be able to *adapt* to the conditions of its operating environment and ensure that its performance never becomes worse than some minimal threshold. In this case, that threshold is the MFT when jobs are all run locally.

4.5 Effect of evaluation delay

The evaluation delay is the amount of time an agent waits after sending out a request for bids before it evaluates the bids which have arrived. One issue that must be dealt with is what the evaluation delay should be set to.

It seems that the evaluation delay should be related to the message delay, because this effects how long it will take for bids to arrive once an agent has sent out a request for bids message. If the evaluation delay is less than 2 times the message delay, then an agent will never receive any bids from remote agents by the time it evaluates bids, meaning that at most it will have a bid from itself, but possibly not, because it may be

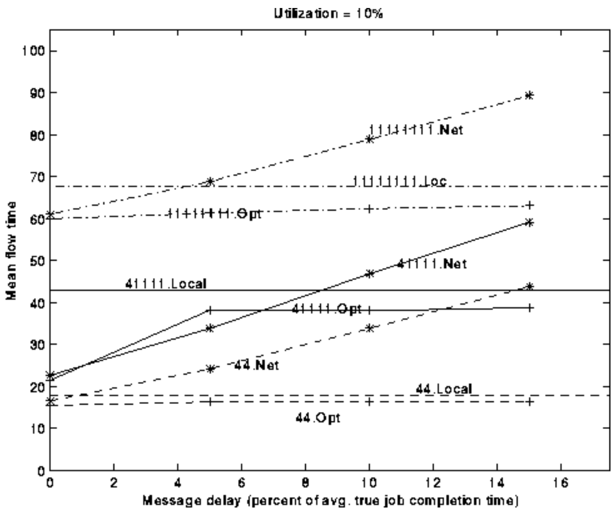

Figure 3: Effect on MFT of increasing message delay. Note that in the Opt setup the MFT never exceeds that of the Loc setup for all three configurations. Utilization is 10 percent.

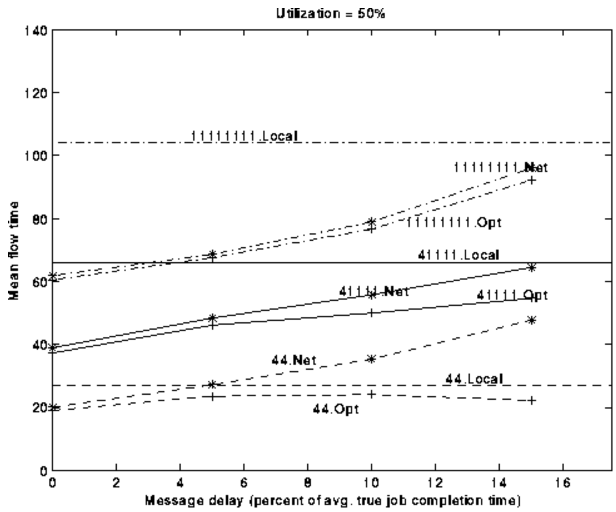

Figure 4: Effect on MFT of increasing message delay. Note that in the Opt setup the MFT never exceeds that of the Loc setup for all three configurations. Utilization is 50 percent.

busy executing a job. In this case, the agent switches to a mode where it will accept the next bid that arrives for the job. So should the evaluation delay be set to at least 2 times the average message delay, in order to allow remote bids the chance to arrive by the time the agent evaluates bids?

From our simulations, it appears that it is almost always best to set the evaluation delay as low as possible, no matter what the operating conditions are (system utilization or message delay). This is why in the simulations for Figure 3 and 4, the evaluation delay was always set to 1. Doing so also eliminates the problem of setting the evaluation delay as a function of message delay, given that it may be difficult to estimate the message delay a priori, and also that the message delay may change over time.

Figure 5 shows a three-dimensional surface plot, showing MFT as a function of both system utilization, and evaluation delay as a multiple of message delay. The configuration was 41111 and the message delay was set to 5 percent of the average job length (60). We can see that under most operating regimes the MFT is lowest for the smallest evaluation delay. The exception to this is for low system loads (0.3 and under), when setting the evaluation delay to greater than the magic 2 times the message delay yields slightly better performance. We found that the general shape of Figure 5 holds for a wide range of configurations and message delays.

5 Adding Learning

We have thus far described only the base agent behavior. We now describe the addition of learning behaviors to the agents that allow them to perform better under a wider range of operating conditions.

5.1 Dealing with Message Delays

The first way in which Challenger agents learn concerns message delays. Given modern networks, message delay is likely to be pretty small. Given modern computers, though, job completion times can be pretty small too, and we have shown in Figures 3 and 4 that once message delay exceeds a small fraction of the average true job completion time, system performance degrades significantly.

To help avoid the degradation caused by message delay, Challenger's agents have the ability to learn the level of lag in the network, and use this information to make decisions about job assignment that result in better performance.

All messages in Challenger are globally time-stamped by the sending agent. When a message arrives, the recipient agent calculates the lag. It uses this information to update a table that stores the agent's current estimate of the lag between itself and all other agents in the network. Lags are assumed to be symmetric, i.e. the delay from agent A to agent B is equal to the delay from B to A. (Note: In our simulation we always assume the communication lag from the agent to itself is zero. In a properly implemented Challenger system, this can be practically assured.) The agent estimates the lag to another agent by averaging the past Z lags to that agent. By adjusting the value of Z, one can roughly control the sensitivity of the agent to changing network conditions. A small Z means the agent is more sensitive to changing lags; a larger Z

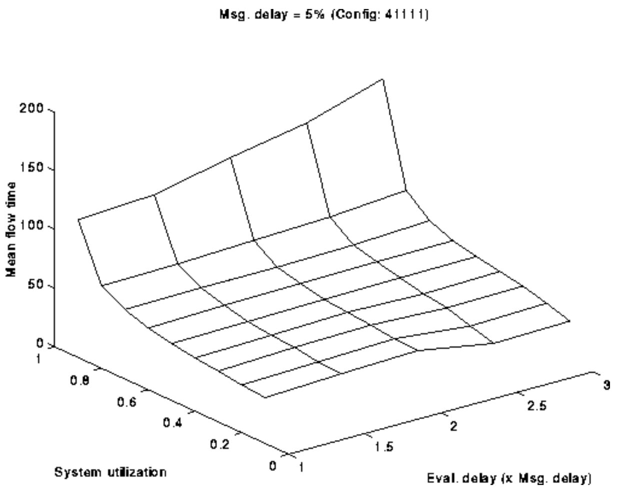

Figure 5: MFT as a function of system utilization and evaluation delay. Evaluation delay is measured as a percentage of message delay. Message delay was set to 5 percent of avg. true job completion time. The configuration was 41111.

implies the agent will be less sensitive.

An agent uses this network lag information to avoid assigning a job remotely when it is expected to be better to run the job locally. When communication lag between agents becomes high (in terms of percentage of average job processing time), overall performance can become really poor, i.e. worse than just shutting down the network and running all jobs locally. To prevent this from happening, the agents have the following behavior:

- When a job is originated, the agent computes its estimate of the minimum remote delay ($EMRD$). The $EMRD$ is an estimate of the minimum amount of time it would take to run a job on a remote agent, and is given by the following formula:

$$EMRD = 4 \times AVGLAG + EVALLAG \tag{2}$$

  $AVGLAG$ is the current average overall network delay, computed by averaging the current estimated lags to all the agents in the network. We multiply $AVGLAG$ by 4 because it takes exactly four messages to run a job remotely: a request for bids message, a bid message, an assignment message, and a result message. $EVALLAG$ is given by the following:

$$EVALLAG = \begin{cases} 0 & \text{if } EVALD < 2 \times AVGLAG \\ EVALD - 2 \times AVGLAG & \text{otherwise} \end{cases} \tag{3}$$

  $EVALD$ is the agent's evaluation delay, i.e. the amount of time an agent waits after sending out a request for bids before it evaluates the bids. $EVALLAG$ is the estimated amount of time that will be spent in evaluation delay *exclusive* of the estimated time spent waiting for messages to arrive.

- If the estimated completion time of the job *on the local processor* is less than $EMRD$ AND the processor is currently idle, then the agent runs the job locally and does not broadcast a request for bids message to the entire network, as is always done in BASE mode.

- If the processor is currently busy executing some job, and the estimated completion time of the job *on the local processor* PLUS the estimated remaining time left of the currently executing job is less than $EMRD$, then the agent runs the job locally and does not broadcast a request for bids message to the entire network.

The heuristic being used is simple: If it is clearly faster to run a just-originated job locally, then do so, and dispense with the usual job assignment protocol. This serves the duel purpose of having the job complete sooner (desirable, given that we are trying to minimize mean flow time), and frees up the rest of the agents in the system from having to make bids and waste precious time waiting for messages to arrive. It might seem that this heuristic is guaranteed to produce better results, but there is a potential downside. When the job is run locally, there might be some remote job that could have benefited (in terms of reducing the time it would take to run) even more than the local agent benefits from running its job in the quickest possible time. This seems unlikely, but it is nonetheless a theoretical possibility. We shall see shortly that this heuristic does indeed produce the desired results.

When the Challenger agents have the aforementioned behaviors activated (which can override the base behaviors), the system is said to be in NTWRK mode.

5.2 Simulations with message delay learning activated

We can now look at Figure 3 and 4 and understand the significance of the lines labeled with Opt. These lines came from simulation runs conducted with the agents in NTWRK mode. We see that there is a significant performance improvement over running Challenger in BASE mode only (indicated by the Net labels), especially with higher message delays. With NTWRK behaviors on, the MFT *never* exceeds the MFT when all jobs are run locally. Challenger is now adaptive enough so that its performance never exceeds the minimal acceptable threshold.

With the addition of the NTWRK mode behaviors, we get the best of both worlds. When the message delay is low, we get the benefit of improved performance due to the standard BASE behaviors. When the message delay is high, performance never becomes worse than the case where all the jobs are run locally. The intuition is straightforward: as message delay increases, more and more of the jobs are run locally, preventing agents from wasting time waiting for messages to arrive. Only very big jobs will trigger request for bids broadcasts, which makes sense, because only these jobs can potentially benefit by being run remotely (in terms of running faster than they would locally).

5.3 Dealing with Estimation Inaccuracy

A second way in which Challenger agents learn deals with estimation inaccuracy. There are two sources of estimation inaccuracy: when the originating agent estimates the job completion time on a benchmark processor (which is used as the job's priority value), and when a bidding agent estimates how long it will take to complete the job. We chose to only deal with the inaccuracy resulting from the bidding agent.

What are possible sources of this inaccuracy? There are potentially many. First, there is the fundamental difficulty in estimating job completion times. Second, a particular bidding agent might be consistently under or over-estimating on its bids, for whatever reason. Perhaps the agent's machine is very heavily loaded (caused by say, the user having ten Web browsers up at once, all running Java). Then, the fact that the unloaded machine is X times as fast as an unloaded baseline machine doesn't translate into it being able to do a job in $\frac{1}{X}$ of the time. Or, an agent's bid might be consistently off because it is being mischievous or malicious.

Our learning scheme is designed to address the second source of inaccuracy given above: agents whose bids are consistently off from the performance they actually deliver. We would like to exploit this; that is, in our bid evaluation process, "penalize" those agents which consistently underestimate job completion times, while "rewarding" those agents which consistently overestimate.

To achieve this, the Challenger agent is endowed with the following simple learning behavior:

- When a job is assigned to the winning bidder, record their bid, i.e. how long they "promise" to take to complete to the job.
- When the job result is returned, compute the ratio of the actual completion time to the "promised" time. Call this ratio R_{a-to-p}. Note that the actual completion time is adjusted to account for network lags. This is done by subtracting 2 times the current estimated lag to the agent which ran the job. The actual completion time is measured from the time the job is assigned to the winning bidder, to the time when the result arrives at the originating agent. Thus, two network delays need to be subtracted out, one for the assignment message and one for the result message.
- Use R_{a-to-p} to update the "inflation factor" for the agent which ran the job. The inflation factor is just the average of the last Y R_{a-to-p}'s for that agent. Putting it another way, the agent "remembers" the recent performance of the other agents. By increasing or decreasing the value of Y, one can adjust the "length" of an agent's memory. There are probably fancier memory decay schemes we could use, but this seemed adequate and was easy to implement.
- During the bid evaluation process, adjust each agent's bid by multiplying it by the agent's current inflation factor. For instance, if an agent has recently been making perfectly accurate bids, its inflation factor will just be 1.0, and its bid will not be altered. On the other hand, if an agent has been recently turning in job completion times that are twice as slow as what it promised, then its bid will be multiplied by an inflation factor of approximately 2.0.

When the Challenger agents have the aforementioned behaviors activated (in addition to the standard BASE mode behaviors) , the system is said to be in AGT mode.

5.4 Simulations with estimation inaccuracy learning activated

Figure 6 shows the results of simulation runs in both AGT and BASE-only mode. The configuration used was 2222 with no message delays. Unlike all the other simulation runs, where the agents always made bids that were perfectly accurate, in this setup one of the agents consistently ran jobs 4 times slower than what it promised in its bids. This models a scenario where one of the machines in the network is extremely heavily loaded, causing it to consistently underestimate its bids. The solid line denotes runs in BASE-only mode (without learning), and the dashed-line denotes runs in AGT mode (with learning). We can see that the performance is clearly better with learning on. Note that the gap in performance between BASE-only mode and AGT mode decreases as system load increases. The reason is that as utilization goes up, agents have less and less bids to choose from at bid evaluation time. Only if the agent has multiple agents to choose from does having information about an agent's performance reliability matter. If an agent is known to consistently underestimate on its bids, but is the only agent available to run the job, then this information doesn't help at all, since the job will be assigned to it regardless.

6 Comparison to Related Work

We now compare Challenger to three other systems for doing distributed processor allocation.

6.1 Malone et al.

Malone's Enterprise is the closest system we know of to Challenger [Mal88]. Its DSP architecture is very similar to the BASE behaviors of Challenger agents.

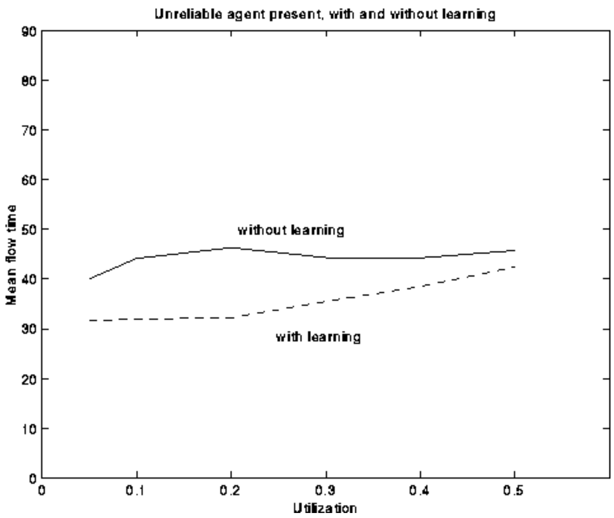

Figure 6: Results of simulations in AGT mode (with learning) and BASE-only mode (without learning). Configuration was 2222, with one of the agents consistently underestimating its bids by a factor of 4. Message delay was zero.

In fact, we were able to duplicate nearly all of Malone's results by running simulations in BASE mode.

The main difference between our work and Malone's is that our agents have learning capabilities whereas Enterprise does not. The Challenger agent's ability to learn about network lags enables it to make decisions about when to run jobs locally or remotely, which allow overall system performance to remain good (better than the base case of running all jobs locally) even when message delays become large. Enterprise's performance under conditions of large message delays, on the other hand, deteriorates dramatically. The Challenger agent's ability to learn about the estimation inaccuracy of bidding agents lets it assess bids in a more accurate manner. This allows the system to perform better than Enterprise in the face of agents which are consistently unreliable.

6.2 Eager et al.

Eager et al. present an algorithm which is fairly representative of distributed heuristic algorithms for processor allocation [Eag86]. The way it works is as follows: A job originates on a machine. If the machine is under-loaded, it runs the job locally. If it is overloaded, it sends out "probes" to other machines in the network, asking if they are under-loaded or overloaded. If a machine comes back and says it is under-loaded, then the job is sent to that machine.

It is difficult to compare this algorithm (which we call Eager) to systems such as Challenger or Enterprise, because some of its basic operating assumptions are different. For example, in Eager processors do time-sharing, i.e. they run more than one job at a time. In the Challenger and Enterprise world views, though, processors are resources which can only be utilized by one job at a time.

Fundamentally, algorithms like Eager suffer from the same shortcomings that Enterprise does: they cannot adapt and thus are not robust under a wide range of conditions. For instance, suppose one of the processors in a network running Eager consistently said that it was under-loaded when in fact it was really overloaded. Other processors would keep sending their jobs to this processor and then wait and wonder why their jobs were taking so long to complete.

6.3 Ferguson et al.

Fegurson et al. present a load-balancing economy based on market principles [Fer88]. They assume network of processors, each with a fixed performance level. They also assume a set of communication links between every processor, each with a fixed delay. Jobs arrive and purchase services from the processors: running time and transmission over their links. Each job attempts to minimize the cost of running itself as well as how long it takes to complete. An auction model is used to determine the going prices for processor services — both the cost of running on a processor, and the cost of transmitting information over the links.

Again, it is hard to compare this work with Challenger, because so many of its basic assumptions are different. We argue that while Fegurson's work is theoretically interesting, it is unclear how it could be translated into a real working system. For one thing, it assumes that operating conditions are completely static: processors have fixed performance and network delays are constant as well. This, as we have argued, is not a realistic assumption. Also, the notion of "price" is vague. How would jobs pay for a processor's services? Would a job be allocated a wad of virtual cash by their creator, the amount depending on how much she values that job's rapid completion? What would the processors do with the money that they earn? These are but a few of the questions that this work poses.

7 Future Work and Conclusion

Our work on Challenger in the near future will focus on conducting more simulations over a wider range of network configurations and conditions (message delays, system utilizations, etc.) to make sure that the results presented here hold up. The next step after that is to build a real Challenger. Our plan is to choose a limited domain of jobs and have Challenger do processor allocation with only those types of jobs. We are considering selecting Java jobs (both compilations and applications) as the domain. We think the platform-independent nature of Java will make it easier to set up a system where jobs can run on any machine in a network. This would not be so easy with a language like C++, say, where platform dependencies abound.

In conclusion, we described Challenger, a service agent for doing distributed processor allocation. Challenger consists of multiple agents, each of which is responsible for the assignment of jobs and allocation of processor resources for a single machine in a network. The base behavior of the Challenger agent is based upon a simple market bidding model. Learning behaviors were added to the agents to have the system perform better under a wider range of operating conditions; namely, in the face of large message delays and agents which make inaccurate bids. These behaviors make Challenger much more robust and adaptive, distinguishing it from other systems for doing distributed processor allocation. We believe Challenger to be an important step towards building agents that make the user's environment a more productive and enjoyable place to be.

References

[Cle96a] Clearwater, S. 1996. *Market-Based Control: A Paradigm for Distributed Resource Allocation.* Ed. Clearwater, S. World Scientific Publishing, Singapore.

[Cle96b] Clearwater, S., Costanza, R., Dixon, M, and Schroeder, B. "Saving Energy using Market-Based Control." 1986. In: Market-Based Control: A Paradigm for Distributed Resource Allocation. Ed. Clearwater, S. World Scientific Publishing, Singapore.

[Eag86] Eager, D.L., Lazowska, E.D., and Zahorjan, J. "Adaptive Load Sharing in Homogeneous Distributed Systems." 1986. IEEE Trans. on Software Engineering, vol. SE-12, pp. 662-675.

[Fer88] Ferguson, D.F., Yemini, Y., and Nikolaou, C. "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems." 1988. In Proceedings of International Conference on Distributed Systems (ICDCS 88). San Jose, California: IEEE Press.

[Mae94] Maes, P. 1994. Agents that Reduce Work and Information Overload. *Communication of the ACM.* Vol. 37, No.7. 31-40.

[Mal88] Malone, T.W., Fikes, R.E., Grant, K.R., and Howard, M.T. "Enterprise: A Market-like Task Scheduler for Distributed Computing Environments". 1988. In: The Ecology of Computation. Ed. Huberman, B.A. Elsevier, Holland.

[Tan92] Tanenbaum, A.S. 1992. *Modern Operating Systems.* Englewood Cliffs, New Jersey: Prentice-Hall.