An Economic CPU-Time Market for D'Agents

Ezra e. k. Cooper, under the direction of Bob Gray

June 2, 2000

Abstract

A usable and efficient resource-management system has been created for use with D'Agents. The software dynamically negotiates a price rate for CPU time, using the competitive bids of mobile agents that offer currency in return for fast computation. The system allows mobile agents to plan their expenditures across many hosts while minimizing the time needed for their tasks. The ability to price CPU time opens the door for service owners to be compensated for the computation consumed by agents and provides an incentive for servers to allow anonymous agents. We discuss the theoretical background which makes a CPU market system possible and the performance of the D'Agents market system.

1 Background

1.1 Mobile Agents as a Paradigm for Distributed Computing

In recent years mobile agents have been proposed as a new approach to distributed computing. In contrast to symmetrical modes of parallel processing, where all processors are given the same program to run and the job is split between them, or multi-threaded systems, where threads can be scheduled on arbitrary individual processors, the mobile-agent paradigm gives the software designer explicit control over the location of execution of each thread of control.

Advantages of this approach are various [9]. The largest is that agent technology allows the (small) program to come to its (big) data across a network. This means that organizations can make large databases available and let users access this information with their own programs, without requiring users to download the entire database in order to do such processing. Present-day WWW search engines make a large body of information available but only through a consumer-oriented search interface; other kinds of queries are impossible. Were these services to adopt agent technology, enterprising and curious individuals could write programs to extract other kinds of information, without needing a copy of the database on their local workstations. Serving more customers' needs would presumably offer greater revenues for organizations with large databases, and at a small expense.

Yet for agent technology to be useful, we need a way to manage anonymous agents' resource usage, particularly CPU time, and a way to compensate agent servers for the use of these resources. A good system of compensation would automatically adapt to changes in demand, rather than requiring the server owner to set a price *a priori*, and it would allow agents to buy quicker execution with more money.

1.2 Dynamic price determination

We value a computational resource by the rate at which a program can use it to take certain kinds of steps. In a good pricing system, the price per unit of computation will be determined automatically, driven by supply and demand.

The model discussed here was developed by Jonathan Bredin as part of his Ph.D. research at Dartmouth College [2, 3]. Here, in return for higher bids, agents are given a greater share of the processor and hence execute faster. In this model, an agent declares a bid function which, given the rate at which other agents are willing to pay, determines the rate at which the agent itself is willing to bid. Larger bids then allow an agent to execute more quickly and smaller ones force it to execute more slowly.

The model [4] used in the D'Agents market can be summarized as follows: an agent i declares to the host a bid function $u_i = g_i(\theta)$; this function returns the rate u_i at which the agent would pay the server in a situation where the total of all agents' pay-rates were θ; it makes this offer in return for $\frac{u_i}{\theta}t$ of every t of that server's scheduling quanta (see the discussion of proportional-share CPU schedulers below). Given this bid, however, other agents will want to change their bids. An agent can either step up its bid to get a good fraction of the CPU, or it can step back, unwilling to pay high rates. The algorithm used in the D'Agents market system finds an equilibrium set of bids where no agent wants to change its bid because of the others.

This bid function trades off expense against speed of execution: for low θ, the agent will offer a bid almost as large as that θ, since the price of computation is cheap and it can get a large share of the processor for this low price. At some critical θ, an agent will want to cut back on spending since the server time is too expensive: it will get more out of spending its money elsewhere. It still wants to finish a job at this server, though, so it offers a small positive bid, in order to execute slowly without wasting too much money. If an agent will only execute at one host, it will be willing to spend all its currency at that host, or just enough to outbid the nearest competitor if it could thereby offer less. On the other hand, if an agent has many tasks, it will want to weigh the relative costs of time at those hosts and bid less on expensive servers in order to save money for cheaper ones where it can save a greater amount of time with the same amount of money.

Bredin derived a bid function that minimizes the total time spent by the agent while not spending any more than the agent's allotted money. A full explication of the result can be found in [4]; some key features are summarized here. The result springs from the following assumptions:

1. Agents know in advance which hosts they will visit,
2. They know, for each host, the amount of processing they will perform there,
3. A server can closely estimate its capacity, or rate of computing.

Indeed, some agent applications might not lend themselves to such estimates and predictions. Still, this bidding system will serve a broad range of applications.

Having collected the bid function of each agent, the server determines the value $\theta = \sum_i u_i$ such that $u_i \approx g_i(\theta)$ for each i. This is the set of satisfactory bids: θ is their sum and each bid is the amount that the corresponding agent would actually be willing to offer if all others were offering those bids.

Now agents can plan their expenditures as follows: Each agent has a remaining reservoir of (electronic) currency I, the value its owner has placed on the agent's mission. It also knows, for each server k that it plans to visit *after* the present server, the going rate θ_k of computation at server k (its 'demand'), the capacity c_k of the server and the size q_k of the task it wants to perform there. It is the agent designer's job to estimate q_k, which is not necessarily easy. Still, the better the designer can approximate the size of the task, the better the results, either in money spent or in time taken. Note that the server demand θ_k is the price of time *received* by any agent, whereas each agent's u_i is the rate at which it will pay against wall time.

Bredin's paper shows that if the agent has this information and wants to minimize the total time taken for all its jobs (which presumably it does), then its optimal bid at server 1 (the next one it will visit) is

$$u = g(\theta) = \frac{(\alpha - \beta\theta)^2}{2\gamma^2}\left(-1 + \sqrt{1 + \frac{4\gamma^2\theta}{(\alpha - \beta\theta)^2}}\right)$$

on the interval $[0,\ \alpha/\beta]$ and 0 outside that interval. Here the coefficients α, β, γ uniquely define the agent's bidding strategy; they refer as follows:

$$\begin{aligned}
\alpha &= I - \sum_{k\neq 1} \frac{q_k}{c_k}\theta_k \\
\beta &= \frac{q_1}{c_1} \\
\gamma &= \sum_{k\neq 1} \frac{q_k}{c_k}\sqrt{\theta_k}
\end{aligned}$$

We can interpret these co-efficients roughly as follows: α is the agent's estimate for how much currency it has available to bid at this server (note that it usually will not bid this much since it wants to bid less if it can, particularly if doing so will allow it to save more time later at a cheaper server). β is the size of the job relative to the capacity of the server: the time the job would take if this agent had the server to itself. Thus α/β is the maximum rate at which the agent can afford to pay for service at this server, in order to have enough money left over to finish its other tasks. At that rate, however, it would need the whole processor; otherwise it would take longer than the ideal time and it would not be able to afford to pay at the rate of α/β for so long. Now, if it *could* get the whole processor, it

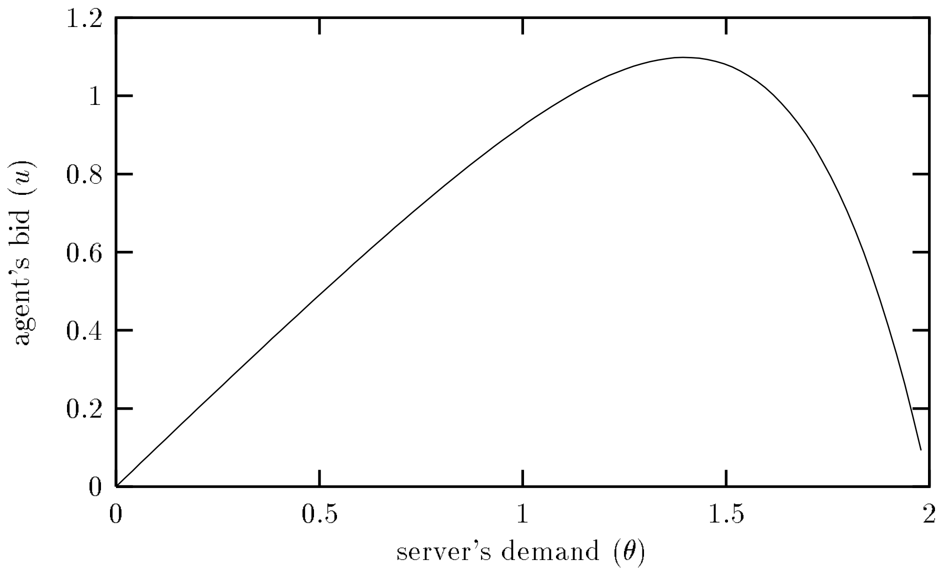

Figure 1: Example plot of agent's bid vs. server load, $\alpha = 100$, $\beta = 50$, $\gamma = 15$.

would only be because no other agents are bidding, in which case it need not pay anything at all. If there is competition, on the other hand, then the agent will necessarily receive only part of the server and must expect to spend more time there. As such it will bid at a lower currency rate: this is why the curve in Figure 1 decreases at the right, even though the 'going rate' θ of the host is less than the agent's maximum pay-rate α/β.

γ is a factor which reflects the agent's reluctance to pay at this server as opposed to other servers. A high γ will discourage the agent from bidding here, because it indicates that other servers will tend to give it a better "bang for its buck." In other words, the agent saves more time by offering an extra dollar at another server than it does by offering an extra dollar at this server. When it has no jobs at other servers, the agent will have $\gamma = 0$, but this creates a discontinuous function, so the D'Agents bid management daemon pins γ values at the low end to $\alpha/1000$. Experimentation showed that this limit does not lead to significant numerical errors and still closely approximates the desired function.

1.3 Proportional-share CPU scheduling

If agents are going to be scheduled according to the amount of money they offer, they want to execute twice as quickly when they offer twice as much money. Yet schedulers in most present-day operating systems offer a 'priority' system as the only way to control the allocation of the CPU: processes with higher priorities are more likely to get a given quantum of time than those with lower priorities. These schedulers usually offer progress guarantees—for example, that each process will necessarily take another step eventually—but make no quantitative offers as to the rates at which these processes will accrue CPU time.

What is needed is a *proportional-share* scheduler, a scheduler which allocates time to processes in the proportion of their *ticket* holdings. If a process holds t tickets during a period when the total number of tickets held on the system is T, then a proportional-share scheduler will give this process t/T of all the scheduler quanta it allocates during that period, within some tolerance known as the *throughput accuracy*.[1]

To support the D'Agents market system, a modified Linux kernel called QLinux was selected. QLinux implements a scheduling algorithm known as Start-time Fair Queuing [5]. The algorithm schedules runnable processes in order of their 'finish tags.' The finish tag of a process is determined by $s + 1/t$ where s is the finish time of the last quantum the process received (or the time when it became runnable, if it was blocked) and t is the number of tickets held by the process.

2 Project Design

A system was implemented whereby agents in the D'Agents system [6] can register at each host a bid function which determines, given the total bid (the demand θ) at that host, the rate u at which the agent would be willing to pay to get cu/θ instructions per second (where c is the number of such instructions that the processor can execute in a second). This system, `bidman`, consists primarily of a daemon running alongside the agent server. The `bidman` daemon collects and manages the bid functions, and whenever the pool of agents (the 'market') changes (whenever one is added or removed, or changes its bid function), the server re-calculates each agent's payment rate using these bid functions. The bid functions are assumed to be of the

[1] When processes block, they get less time than is allotted to them, but the other processes still get fair time: the ratio of quanta received by any two non-blocked processes will be the ratio of the tickets held by those processes.

optimal form discussed above, and are declared only in terms of the three co-efficients α, β and γ. An 'expenses' field associated with each agent, which is available as a variable to the agent program and persists across jumps, is incremented by the amount of currency which the agent has consumed since the last change in the market. In an open system, the agent would eventually be charged this amount via a secure electronic currency system. In the present system, agents are trusted to keep track of their own expenses.

2.1 Organization of Scheduling

The scheduler used by the underlying operating system (QLinux) supports a hierarchy of scheduling 'nodes' where the children of any node are scheduled proportionally with respect to one another, according to their relative ticket holdings. In the D'Agents market system, all agents which have declared a bid are children of a single node; no other processes are part of this node. Agents which have not yet bid reside in a different node which has a small number of tickets. This ensures that, whatever part of the processor is allocated to the agents' node, they share that time in the correct proportion; furthermore, the fraction of the processor devoted to background tasks and to the agent server is limited to a certain constant fraction.

Suppose for the sake of example that the agent-server's scheduling node has 9 tickets, the active-agents node has 90, and the waiting-agents node has 1 ticket (the actual numbers can be determined by the system operator). Then 9% of the CPU will be devoted to server processing, 90% to agent tasks, and agents which have not yet bid have to crawl along, sharing a mere 1% of the CPU between them, until they bid, at which point they'll be moved into the active-agents node. Now if there are two agents in the system, A and B, with 2 and 3 tickets respectively, then A will get 40% of the time devoted to the agent node and B will get 60%; thus A will get $40\% \cdot 90\% = 36\%$ of the CPU.

So, while agents have to share the processor with non-agent processes, it is as if they are running on a slightly slower CPU (in this example, one of 90% the speed of the actual CPU). The "server capacity" value that agents use takes this into account (it is based on a benchmark[2] that is run as an agent). Note that, when the server or other background processes on the host

[2]To define specific benchmarks useful for agent applications is beyond the scope of this project. Here we used a simple integer test which times two loops, one of which performed one more integer operation per iteration than the other. The extra time per iteration spent by the bigger loop is taken as the time to perform one Tcl integer operation.

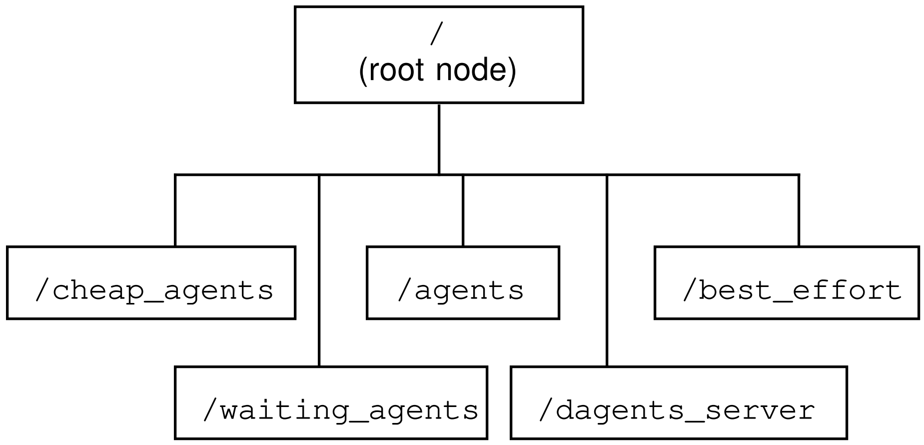

Figure 2: The QLinux scheduling hierarchy used in the D'Agents market system

block for I/O, quanta that would otherwise be given to them will be given to busy agents, and thus agents may complete their jobs slightly earlier than expected. The variability of this allocation is a necessary disadvantage of the system, since system upkeep requires an unpredictable, if small, amount of processing. Suppose for the sake of example that 90% of the CPU is allotted to agents. As long as non-agent processes are processing busily, agents will share amongst themselves that 90% of the CPU. But when most non-agent processes are blocked (a frequent occurence), the scheduler gives extra time to the busy agent processes, rather than let the processor go to waste. Ideally, future OSes will offer precise accounting for user processes and make this information available to a secure super-user process (in fact, it would only require small modifications to the existing QLinux kernel). QLinux does not offer this kind of accounting so the present D'Agents market system gives the benefit of the fluctuation to the agent's wallet. Presently, the market server and the agent agree on a price per time unit and the agent is charged at that rate, against real time, until another auction takes place.

Figure 2 shows the scheduling organization used by the D'Agents market system. All nodes are children of the root node, and its immediate children are proportionally-scheduled with respect to one another. Time thereby allotted to each node is divided among that node's children by proportional-share scheduling as well. The `/agents` node is where active agents reside; it should get the lion's share of the tickets at this level. Agents which have just arrived and have not yet declared a bid reside in the `/waiting_agents` node, which gets much less time. When agents run out of money, they are put in the `/cheap_agents` node, which gets even fewer tickets than the `/waiting_agents` node. The `bidman` daemon and the `agentd` server are placed under the `/dagents_server` node, which gets a moderate allocation of tickets. Finally, the `/best_effort` node is the default where other processes are born, for example, those executed at the command line. Note that the exact ticket holdings of these nodes are an operational parameter rather than a software-design issue and can be adjusted by the operator to suit the circumstances.

2.2 Planning

To help agent designers plan their bids, library functions have been created, following the analysis in Bredin's paper [4]. One routine (`bid_strategy`) takes information about the agent's itinerary (which hosts are to be visited and the size of the tasks to be performed at each) and produces the coefficients that the agent gives to the `agent_bid` call,[3] which takes α, β and γ as arguments and passes them to the bid manager as described below. The model does presume that agents know in advance which servers they want to visit. It further assumes that the agents can fairly accurately assess the size of their tasks in units of total number of (abstract) instructions. See section 4 for some discussion of this possibility. Even agents that have only one task to perform and don't care where they perform it can use the bidding interface; they can simply offer an itinerary of one host and will bid all their endowment at that one host (or less if there is less demand).

Figure 3 gives example AgentTcl code for an agent that declares a bid. The capacity values are hard-coded in this example but ultimately, benchmarks should be devised and agents can be provided which will respond to queries with their servers' capacities. `hosts_to_loads` is another utility routine; it determines the load (demand) of each host in the given list (by querying an agent on that host called `deman`) and it returns a list of those loads.

2.3 `bidman`'s Messages

The bid-management program (called `bidman`) runs as a daemon process separate from the agent server proper and runs with super-user privileges (since QLinux allows only the super-user to modify the scheduling hierarchy). It opens up a System V IPC message queue and listens thereon for

[3] So far implemented only in AgentTcl.

```
#!/usr/agenttcl/bin/agent

package require libbid.tcl

set endwmt 1200
set hosts {happy doc sneezy snow-white doc}

    # these are the sizes of the tasks to be performed
    # at the five hosts above, respectively.
set task_sizes {50 30 15 25 10}
set demands [hosts_to_loads $hosts]
    # capacities are always measured against a particular
    #   benchmark; these are all measured against the same.
set capacities {.01 .035 .012 .02 .035}

set expenses 0
set serv_num 0
foreach server $hosts {
    agent_jump $server    # now expenses is updated to show
                          # amt. spent at prev. host.

    set next_bid [bid_strategy $task_sizes $demands \
                               $capacities \
                               [expr $endwmt - $expenses] \
                               $serv_num]
    agent_bid [lindex $next_bid 0] \
              [lindex $next_bid 1] \
              [lindex $next_bid 2]

    # do work here

    incr serv_num
}
   # finally, we need to get the last bill
set expenses [expr $expenses + [agent_bill]]
```

Figure 3: A Tcl agent which bids.

messages from agent processes. The messages come in five flavors. Those that can be sent by an agent allow it to 1) declare its existence, 2) declare its impending death, 3) set its bid function, and 4) to have its currency consumption (its "bill") reported. The fifth tells `bidman` to make sure the scheduling hierarchy is in place and that all the server processes are in the right scheduling nodes; it is sent by the agent server when it comes online, thus moving itself into the `/dagents_server` node.

A set-bid-function message declares an agent's bid function by carrying the three co-efficients α, β and γ. Its receipt causes the `bidman` daemon to take the following steps: first, it updates its notion of how much time and currency each agent has used. Then it quickly recalculates all the agents' payment rates, taking time linear in the number of competing agents.[4] Finally, it sets each process's ticket holdings, at which time they begin consuming time at the agreed-upon rate.

Upon receipt of a get-bill message, `bidman` calculates a currency amount that approximates the agent's consumption-to-date on the present server; it does this simply by multiplying the time since the last such message by the currency-rate that had been agreed upon at the last auction (Remember, though, that the agreed-upon CPU fraction may not have been what was received by the agent, in which case the agent spends less money and completes sooner). The bid manager daemon keeps track only of the amount of currency charged to the agent for its work *on that server*, so just before the agent jumps to a new host, it gets this bill from `bidman` and adds it into its own C-level 'expenses' variable, which holds a running expenses total for the life of the agent. The `agent_jump` command writes the C-level 'expenses' variable to its counterpart variable in the agent language, so immediately after a jump the agent's expenses, up through the previous host, are available to the agent designer. Each agent at its arrival or inception opens a SysV message queue of its own, which it uses to receive the size of the bill from `bidman`.

The agent's departure or untimely death is signalled to `bidman` with a process-died message. This is sent by the agent itself before departure or, in the case of a crash, by the server's background process which is waiting to clean up after the agent.

Finally, the new-agent message puts the agent into the waiting-agent scheduling node. It will be moved to the active-agents node the first time it

[4] This is done with a bisection search which looks for the fixed point of the function defined by $\sum_i g_i(\theta)$, where $g_i(\theta)$ is the bid function of agent i. This fixed point is the θ for which $\theta = \sum_i g_i(\theta)$, our criterion for a satisfactory set of bids.

bids at this host; an agent should bid immediately after arriving at its host. The waiting-agent scheduling node is intended to limit the processing time of nodes which have not bid yet. Of course, if they never bid, they still have the opportunity to run, albeit slowly (otherwise, they would never get the chance to bid). A more secure system might give agents a timeout: if they have not bid within some period after their arrival, they are terminated or sent back to the source, under the assumption that they were trying to get some free computing time at a low rate.

The only two messages over which agents have control are the message to set the bid function and the message to get the bill—these are sent by the `agent_bid` and `agent_bill` agent commands, respectively. The others (the agent arrival and agent death messages) are automatically issued by the agent interpreters. This way, the agent always gets moved to the `/waiting_agents` node, without the agent program taking any action, so processing is necessarily limited before the agent bids.

3 Tests

One measurement of success of a market system like this is the relationship between agent expenditures and rates of execution. The question is: Do agents which pay dearly get their tasks done sooner? Do agents which are stingy take longer to execute? Figure 4 shows the relationship of average execution rate (total number of instructions divided by total duration) to expense rate (endowment divided by total duration). This data was taken in one run of 120 agents, released at random intervals between 0 and 60 seconds, where each agent's itinerary consisted of hops back and forth between two similar machines (both original Pentiums at 133MHz, with 32 M of physical memory and 56 M swap space). Each agent had a random task size at each host; the task entailed ten thousand integer operations for each unit of task size (the "task" is shown in Figure 5).

The plot shows, on the y-axis, the total of all an agent's task sizes divided by the total time taken (note that no account is made for the network latency, which was a local 10Base-T and had a relatively low latency). The x-axis of the plot indicates the endowment of the agent divided by the time it took. Thus agents which spent their currency more quickly will appear toward the right side of the graph. The plot supports a positive relationship between performance and endowment, and for poorer agents the relationship appears close to linear. For the rarer, better-endowed agents, their extra cash does not necessarily help much; this is because, to get good performance, they

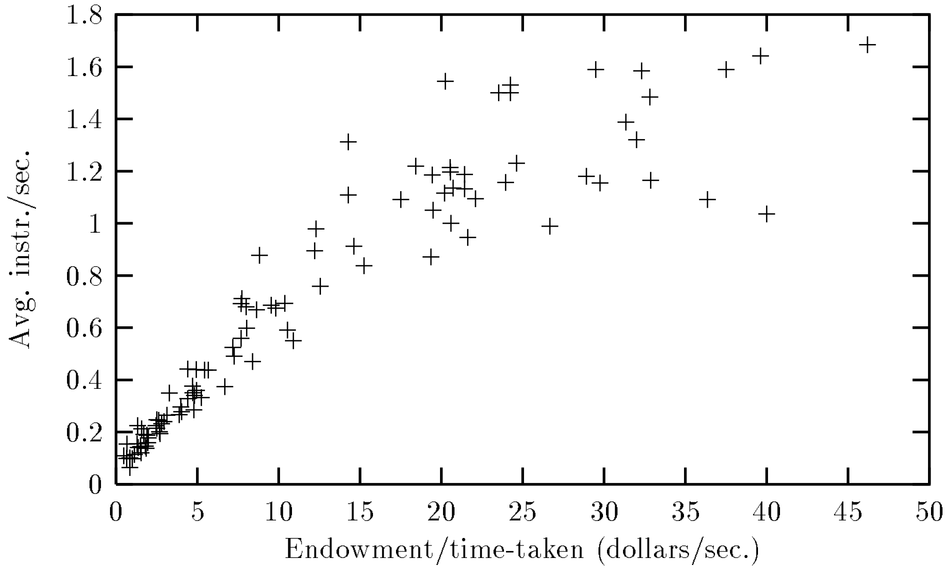

Figure 4: Graph of agent performance vs. rate of expenditure over complete itineraries.

```
set i 0
while {$i < 10000} {
    set job 0
    while {$job < $jobsize} {
        incr job
    }
    incr i
}
```

Figure 5: The Tcl code for the “task” performed by test agents in Figure 4.

only need to outbid their nearest competitors, who are much poorer. As such, we see that the graph flattens out considerably for agents with bigger endowments.

A possible site of inefficiency in a system like this is the time between an agent's arrival at a host and the moment it finishes its bidding process and can begin work with the appropriate number of scheduling tickets. In the present D'Agents market system, this lag time is determined by the number of tickets allotted to the `/waiting_agents` node. However, when the number of tickets allotted to this node is large, we expect more fluctuation in the fraction of the CPU received by agents.

The wait-time before bidding is tested as follows: the agent interpreter stamps a log file when the agent first arrives at a host, and then again when the `agent_bid` call returns. `bidman` stamps the file when it has completed an auction. The time between the arrival stamp and the first auction completed after `agent_bid` returned is taken as the agent's wait time. In a test of 55 arrivals at one host, where 10% of the CPU was allocated to waiting agents, the average wait time was 0.93 seconds.

During the same run, the error of `bidman`'s time-consumed estimates was tested. 20% of the CPU was allocated to non-agent processes, and so the fluctuation in the amount of time given to agents was limited to 20%. That is, agents could share anywhere between 80% and 100% of the CPU in a given time period, depending on the computational demands of non-agent processes. The test showed that `bidman`'s error in estimates of time consumed was about 17%. The error is determined as follows: just before departing, the agent logs to a file the fraction of the CPU it received at that host.[5] When `bidman` is notified of the agent's death, it logs its own estimate of what CPU fraction the agent would have received, had there been no fluctuation. After the run, the two values are compared for each agent, and all the average of these errors is found. This average, 17%, is less than what can be expected due the fluctuation of the CPU fraction shared by agents, which was limited to 20% in this test. This evidence supports the claim that the error in time-consumed estimates is primarily due to this fluctuation.

[5]The agent has access to this information through a standard UNIX system call, `getrusage`, but the bid manager does not have the ability to get this information for processes other than itself. Communication back and forth between agents and `bidman` would be prohibitive, since agents would need to be interrupted in the middle of their tasks. For these reasons, a design decision was made, that `bidman` would use estimates rather than query the agent to find out how much time it consumed.

4 Future Work

This project is the beginning of a versatile, widely usable open system: many features could be added to enhance it.

To improve accuracy, the kernel should be extended to offer process accounting information to superuser processes like `bidman`. This would be a simple modification: the existing `getrusage`(2) system call is just an interface to a deeper routine that takes as its parameter a pointer to the process structure for which accounting should be reported. A new system call could be created that takes a process ID as a parameter, finds the corresponding process structure, and calls the underlying routine. Making this change would improve accuracy significantly: it would mean that `bidman` could determine agents' consumption down to a quanta rather than merely within 20%, or whatever fraction of the CPU is allocated to non-agent processes.

Network bandwidth and other resources could also be managed and priced by the same scheme used here for CPU time. QLinux already provides the infrastructure for managing network flows using HSFQ; all that is needed is an agent interface to it and a way of knowing how much network traffic an agent will produce (or receive). The same pricing model could be used: "task sizes" would become message sizes (or the size of groups of messages), and "server capacities" would be the bandwidth of the server's network connection (some allowances would have to be made for the variability of end-to-end network bandwidth).

Finally, a system like this will be most useful in the greatest number of applications if a system can be contrived for estimating job sizes programmatically. We envision a kind of pre-profiling: an algorithm that could, for a certain modest class of input programs, process the input source code and determine how many of various kinds of instrucions will be needed. While this may seem like pie in the sky, it is not completely un-reasonable that such a technique could be invented; the biggest problem comes with open-ended loops, ones whose limit is not known even while the loop is running (such as a loop that counts the number of elements in a linked list). About these open-ended loops, nothing can be done. But such a system, if it does exist, would be able to handle situations where the limit of a loop *is* known, if only at run-time, because the agent can split up its tasks at each server. So, just before entering a loop, the agent could use its information about the size of the loop to execute another `agent_bid` command to ration its remaining currency on this job.

5 Acknowledgements

Thanks to Robert Gray for advising me on this project, to Jon Bredin for doing all the hard work (the theory), and for having a cool haircut, and thanks to Daniela Rus and David Kotz for being the committe that evaluates this. I am indebted, too, to the residents of Foley House for supporting me in my midnight angst and to the spring breezes for giving me a reason to persevere.

References

[1] Ezra E. K. Cooper, Robert S. Gray. An Economic CPU-Time Market for D'Agents. Senior Honors Thesis, Dartmouth College, June 2000. Available as Dartmouth Computer Science Technical Report TR2000-375.

[2] Jonathan Bredin, David Kotz, and Daniela Rus. Market-based resource control for mobile agents. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 197–204, Minneapolis, MN, May 1998. ACM Press.

[3] Jonathan Bredin and David Kotz and Daniela Rus. Mobile-Agent Planning in a Market-Oriented Environment. Technical Report PCS-TR99-345, Dept. of Computer Science, Dartmouth College, May 1999. Revision 1 of May 20, 1999.

[4] Jonathan Bredin, Rajiv T. Maheswaran, Çagri Imer, Tamer Başar, David Kotz, and Daniela Rus. A Game-Theoretic Formulation of Multi-Agent Resource Allocation. In *Proceedings of the Fourth International Conference on Autonomous Agents*, June 2000.

[5] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems, in *Proc. 2nd OSDI Symposium*, pp. 107–121, October 1996.

[6] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. PhD. thesis, Dartmouth College, June 1997. Avalable as Dartmouth Computer Science Technical Report TR98-327.

[7] Robert S. Gray. Agent Tcl: Alpha Release 1.1. Dartmouth College, 1995. Available at `http://agent.cs.dartmouth.edu/`.

[8] Robert S. Gray. Installing D'Agents: Release 2.0. Dartmouth College, 1998. Available at `http://agent.cs.dartmouth.edu/`.

[9] David Kotz, Robert S. Gray. Mobile Code: The Future of the Internet. In *Proceedings of the Workshop "Mobile Agents in the Context of Competition and Cooperation (MAC3)" at Autonomous Agents '99*s, pages 6–12, May, 1999.

[10] C. A. Waldspurger. "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," Ph.D. thesis, MIT, 1995.

A `bidman` User's Manual

A.1 Introduction

This documentation extends the documentation for AgentTcl [6], which is necessary background reading for this documentation.

`bidman` is an optional extension of the D'Agents mobile agent system that allows economically-based resource management including the dynamic calculation of prices for such resources based on supply and demand. The only resource controlled by the present `bidman` system is CPU time: agents plan their computational tasks and ration their currency over the extent of these tasks, executing more or less quickly depending on their rate of expenditure relative to other agents. As of this writing, the bidding system interfaces only with AgentTcl and not with the other agent languages in D'Agents.

For more information on the background and performance of the system, see [1].

A.2 Bidding model

A full explanation of the bidding model employed by `bidman` can best be got from the paper by Jon Bredin that introduced the model [4]. However, a few points will be worth noting here.

Each agent declares a bidding function, whose input is the total of all bids at the present host and whose output is the agent's own bid. The server will find a set of bids for which each agent's bidding function returns the agent's bid in that set: that is, it is a set of bids with which all agents are happy. Each agent declares, through its bid function, how much of the server's load it would make up, if the entire community of agents at that server were paying at the given rate.

An agent's optimal bidding function is characterized by three real-valued parameters, commonly referred to as α, β, and γ. The first expresses the amount of currency the agent has available for its task at this server; the second expresses how much time it would take if the agent had the server to itself. Thus α/β is the maximum possible rate at which it could pay for service. As such, the bidding function is positive in the range $(0, \alpha/\beta)$ and is always $\leq \alpha/\beta$. Agents almost never bid that much, however, since by offering a bit less they can often save a lot of money while losing only a little time; the time can then be caught up at a cheaper server. An example bidding function is shown in Figure 1.

In return for a bid of u, an agent expects to receive u/θ of the CPU, within some tolerance. To illustrate: if one agent is bidding at a rate of \$0.50/sec. and another is bidding at \$0.75/sec., the total bid is $\theta =$ \$1.25/sec. The first agent will receive 2/5 of the CPU (that is, two seconds out of every five seconds of clock time) and the other will receive 3/5 (three seconds out of every five seconds of clock time).

A.3 Installing

`bidman` requires QLinux, a modified version of the Linux 2.2.0 kernel. QLinux extends the Linux process scheduler to allow hierarchical proportional-share scheduling, which allows `bidman` precise control over processes' rates of execution. For more information about proportional-share scheduling and the algorithm implemented in QLinux, see [5]. The QLinux kernel itself is available at `http://www.cs.umass.edu/~lass/software/qlinux/index.html`. The development team included Pawan Goyal (Ensim Corporation, formerly with AT & T Research), Jasleen Kaur Sahni (Univ. of Texas), Prashant Shenoy (Univ. of Massachusetts), Raghav Srinivasan (Univ. of Massachusetts), Harrick Vin (Univ. of Texas), and T R. Vishwanath (Univ. of Texas).

With QLinux installed, the next step is to install D'Agents. This can be done by following the directions in the D'Agents installation documentation [7, 8]. The market system is an optional package available alongside the central D'Agents package.

The bid management daemon (`bidman`) needs to have super-user privileges in order to have access to the scheduler. It can be run by hand from a root login shell, but adding it to the system startup scripts is recommended; this way it is always running and needs no extra attention.

A.3.1 Adding `bidman` to your system's startup scripts.

On my development system (a SlackWare Linux distribution with the QLinux kernel added), this was easy; hopefully this example will serve to illustrate how it can be done on other installations.

First, I looked in `/etc/inittab` and saw that, on going to user level 3 (the usual multi-user mode), the system runs the `/etc/rc.d/rc.M` initialization script. Careful inspection of this file revealed that it executed another, `/etc/rc.d/rc.local`, which is meant to be modified by the system administrator. It turned out to be empty, so I added the command

```
cd /usr/agenttcl/server/bidman; ./bidman -q
```

The `-q` flag indicates quiet mode and instructs `bidman` not to print status updates to the terminal.

A.4 Agent-language commands

Two commands have been added to the agent languages, namely `agent_bid` and `agent_bill`. These commands are discussed in turn below.

`agent_bid` α β γ [<*expenses*>]

This command declares the agent's bidding function to the `bidman` daemon. α, β and γ are the three co-efficients that determine the bidding function (see section A.2). The optional <*expenses*> parameter is the name of the variable where the agent's total of incurred expenses will be stored.[6] During an `agent_jump` command, the interpreter will query the `bidman` daemon for the agent's total bill for the agent's stay *on that server* and will *add* this bill to the value in the expenses variable.[7] These co-efficients will be re-used at each arrival and departure event until the agent calls `agent_bid` again. However, they are updated, at each event, as follows:

- `bidman` will deduct from β the amount of time consumed by the agent. If this adjusted value should fall below the (very small) minimum, a 'safe' value will be used. The 'safe beta' used in this release is the length of the 'bidding period': the average

[6] In the present release, this parameter is ignored and the name `expenses` is always used.

[7] In fact, the variable's value is cached upon arrival at a host and it is this *cached* value that will be incremented at the time of the jump. So writes to this variable by the agent program are lost.

time between arrival/departure events. This way the agent bids as if it were going to run for that much longer. It may run for more (or even less), but as long as another auction occurs within that time span, the agent gets to re-bid, and will ration out its remaining currency for yet another average-bidding-period. This safety feature helps prevent errors in task size from completely bankrupting agents.

- The effective α used is the declared one minus all the agent's expenses to date; the daemon keeps track of these expenses and updates the total before each auction. The amount of time *elapsed* since the last auction is multiplied by the pay rate (the value the agent's bid function had returned at the last auction), and this product is the increment in expenses. Thus expenses are updated at the end of every stable period (where prices and consumption rates are constant) in the history of the server. If this effective α should drop below a small positive value, the agent necessarily offers nothing, and is relegated to a slow-moving scheduler node (`/cheap_agents`).

`agent_bill`

This command takes no parameters and returns the amount of currency the agent has consumed *so far* on *this server*. The returned value is just a reflection of the daemon's internal concept of how much the agent has been charged: it is identical with the amount that is subtracted from the declared α before performing an auction. This routine allows the agent to keep track of its spending and perhaps to perform other activities depending on its expenses.

There is one situation where it *must* be used: to get the last bill at the last server the agent visits (i.e., when it will not call `agent_jump` again) to update the 'expenses' variable a final time.

A.5 Internals

This section is intended for programmers who are extending or modifying the `bidman` software. It explains the internal workings of `bidman` in hopes that future generations of OS hackers will improve upon it.

A.5.1 `bidman` structure

The `bidman` daemon basically does only one thing: it waits in a loop for incoming messages (described below). Some messages cause it to perform an auction, an iterative algorithm nestled within the class `CMarket`. The `CMarket` class also keeps a list of all the agents that are competitively bidding at the server.

A `CMarket` object contains a list of bidders (kept by a `CBidderList` object) and various parameters, such as its demand value (the total of all agent bids—also the price of CPU time *received* by any agent). It supports operations such as: adding bidders, removing them, providing information about a given bidder, and calling an arbitrary function for each bidder. Adding and removing bidders automatically forces an auction.

To perform an auction, we find the fixed point of the function $\sum_i g_i(\theta)$, where g_i is the bid function of agent i. This fixed point is found using a bisection search until the input and output of the function are within the constant `CMarket::TOLERANCE`. The 'error' of this search is the amount by which the input and output differ—it is reported with the auction results and may sometimes be greater than `CMarket::TOLERANCE`, because the fixed-point search gives up after a certain number of iterations. This can happen when an agent has given a bid function that is 'close' to discontinuous, as when γ gets very small.[8] However, γ is pinned to $\alpha/1000$ so that bid functions won't break up from numerical errors.

The range of the bisection search is the continuum between 0 and the largest α/β that an agent has declared. We need to know this largest value before doing an auction. Right now the software just searches through all the agents to find it, in the function `FindMaxAtoBRatio()`.

Note that the `find_fixed_point` function is designed to be modular and usable even outside this project. As such, it takes a C function parameter and can't deal with a class member function. To get around this, we use the old kludge of giving it a C glue function that takes a pointer to an object—the object on which to call the desired member function.

The `bidder` class keeps track of an agent's bill, the relative rate at which it is supposed to be accruing CPU time, and the bidding function, among other things. The `bidder::Stamp()` routine updates the recorded amount of time used, based on the time elapsed and the fraction of that time that was (supposed to be) used by the agent. It can be called frequently to keep the agent's time-used values up-to-date; it *must* be called just before an agent's ticket values change, or else successive stamps will be inaccurate.

[8] The limit of the bid function as γ goes to zero is, in fact, a discontinuous function.

A.5.2 Agent-`bidman` interface

`bidman` creates a message queue when it starts up. It uses the file

`/usr/agenttcl/access/bidman.queue`

as a public reference point for the locating the message queue. Agent processes know to look for this file and to use it as an argument to the `ftok`(3) standard library routine, which returns a key for the queue. This way all processes can easily find the queue and send messages to `bidman`.

To interact with `bidman`, use the routines that send the various kinds of messages (they can be found in the file `generic/genBidding.c`. Each message type has a stub routine that can be used to easily send messages to `bidman`. The messages are discussed below.

NEW_PROC_MSG This message is sent by the AGENT constructor and it includes the process ID of the sending process; when `bidman` receives one of these messages it moves the named process into the slow-moving `/waiting_agents` bucket so that it won't execute too quickly without a bid. The agent is *not* added to the `CBidderList` and will not compete in auctions until a `SET_BID_FUNC` message is received for the agent.

BID_FUNC_MSG This is the message that establishes the agent's bidding function. It carries with it a process ID and the three co-efficients, α, β and γ. The receipt of one of these messages is what causes `bidman` to add the agent to the `CMarket` data structure (unless it is already there, in which case the existing record will be updated). The message also updates agent's bills and causes an auction, the results of which are reported to `bidman`'s terminal.

DIE_PROC_MSG When an agent reaches it's `agent_end` command, or when it exits, it sends this message to `bidman`. It is also sent by the agent interpreter's background handler if it detects that the agent has died abnormally. Thus, whenever the process ceases to exist, this message should be sent; it will remove the agent from `bidman`'s data structure.

GET_BILL_MSG Sending this message with a process ID causes `bidman` to calculate that process's bill and to send it to a message queue that is keyed to the pid. Specifically, the reply queue is identified by a key constructed as follows: the high-order word is `0x6167` (`'ag'`) and the low-order word is the process ID. Thus the agent process can wait on this queue for its bill. Such billing replies can be sent in other situations, however, so agent processes should always look for the last message in the queue to ensure that their bills are up-to-date.

HUP_SCHD_MSG Sending this message causes `bidman` to re-establish the scheduling hierarchy it wants. This means creating the five top-level scheduling nodes, if necessary, and setting their tickets, but it also means moving the server processes into the `/dagents_server` node. `agentd` sends this message on startup so that it will be moved to the proper node even if (as is usually the case) `bidman` was started before `agentd`.