The Ecology of Computation
B.A. Huberman (editor)
© Elsevier Science Publishers B.V. (North-Holland), 1988

Incentive Engineering for Computational Resource Management

K. Eric Drexler
MIT Artificial Intelligence Laboratory,
545 Technology Square, Cambridge, MA 02139*

Mark S. Miller
Xerox Palo Alto Research Center,
3333 Coyote Hill Road, Palo Alto, CA 94304

Agoric computation [I,II] will require market-compatible mechanisms for the allocation of processor time and storage space. Recasting processor scheduling as an auction process yields a flexible priority system. Recasting storage management as a system of decentralized market negotiations yields a distributed garbage collection algorithm able to collect unreferenced loops that cross trust boundaries. Algorithms that manage processor time and storage in ways that enable both conventional computation and market-based decision making will be useful in establishing agoric systems: they lie at the boundary between design and evolution. We describe such algorithms in some detail.

1. Introduction

In the agoric model of computation [II], market mechanisms—prices and negotiations—coordinate the activity of objects. As in existing markets, these mechanisms will allocate resources in a decentralized fashion, guided by local knowledge and decisions rather than by central planning and direction. Through selective rewards, markets will encourage the evolution of useful and efficient objects.

This model raises questions at several levels, ranging from foundations (how can hardware and software support secure market mechanisms?) to high-level emergent properties (how will evolved computational markets behave?). Here we focus on a question at an intermediate level, asking how the basic computational resources of processor capacity and data storage can be managed in a market-compatible fashion, given suitable foundations. We first examine objectives and constraints for decentralized resource management, and then describe a promising set of *initial market strategies*. These are algorithms that can help seed a system

* Visiting Scholar, Stanford University. Box 60775, Palo Alto, CA 94306

with a workable initial set of objects. Initial market strategies (or *initial strategies,* for short), must be compatible with current programming practice and (when widely used) must provide an environment in which resources typically have prices that reflect costs—that is, an environment in which other objects can make economic decisions that make sense.

1.1. The role of initial market strategies

Agoric systems will evolve from lesser to greater complexity. To begin evolving, they will need a comparatively simple structure from which to build. The role of initial market strategies is to serve as a sort of scaffolding: they must support the right things and have the right general shape, but they must also be replaceable as construction proceeds. Though they must embody sound engineering principles, they need not themselves be architectural wonders.

For computation to occur, storage and processing power must be allocated and managed—functions that existing algorithms handle [1,2]. To get a computational market started, we will need initial strategies that also enable market-based decision making. To enable conventional computation, the use of an initial strategy by a set of objects must ensure the reliable scheduling of processes and allocation and deallocation of storage. To enable globally-sensible, market-based decision making, the use of an initial strategy by a set of objects must provide an environment in which the *price* of using an object reflects the real *cost* of that use.

These initial market strategies, however, *cannot* use price information to guide their own actions. To behave reliably in a conventional computational sense, they must pursue their goals regardless of cost. If resource prices are to make economic sense, however, objects at some level (perhaps contracting with systems of initial-strategy objects) must be willing to forgo or delay actions which will cost more than their estimated worth. Initial market strategies exist to provide an environment in which more advanced market strategies can function and evolve (even evolving to replace the initial strategies themselves).

Initial programming environments can provide initial market strategies as default parts of object definitions. Though initial strategies will perform functions ordinarily handled by an operating system or language environment, they need not be foundational or uniform across a system. The foundational mechanisms of agoric systems will delegate decisions regarding processor scheduling and storage management to unprivileged objects, enabling them to follow diverse policies. Initial strategies will simply provide policies that work, until better policies emerge.

The initial market strategies described here are intended to serve two purposes: first, to provide a proof-of-concept (or at least evidence-of-concept) for the feasibility of decentralized resource management meeting the constraints we describe; and second, to provide a point of departure for further work—a set of ideas to criticize and improve upon. For several of our choices, straightforward alternatives can likely be found. Many will prove superior.

Throughout the present discussion, "object" should be considered a scale-independent notion: an object should frequently be regarded as a large, running program, such as an expert system or on-line database. Large objects may or may not be themselves composed of objects, and objects in general need not incorporate any notion of class hierarchy or inheritance.

Some of the following algorithms have relatively high overhead and are not proposed for use with small objects; large objects might use conventional algorithms for fine-grained internal resource management.

1.2. Auction-based processor scheduling

Most objects will need only a fraction of the service of a processor, hence we expect rental to emerge as a major means of acquiring processor time. Since objects will frequently be able to trade off processor use against storage requirements, communications use, or service quality, processor time will have a price relative to these other resources. This price will vary from processor to processor and from moment to moment. If an agoric system is open, extensible, and uses real currency, and if machine owners are alert, then the long-term average price of processor time in the system will reflect the external market price of adding more processors to the system—if it were much higher, the owners could profit by adding processors; if it were much lower, they could profit by removing and selling them.

Since the demand for processor time is apt to fluctuate rapidly, proper incentives will require rapidly fluctuating prices. This can be arranged by auctioning each slice of processor time to the highest-bidding process. The urgency of a task can be reflected in the size of its bid. Auctions can also be used to schedule other resources allocated on a time-slice by time-slice basis, such as communication channels. Again, fluctuating prices can provide incentives for delaying less urgent tasks, leveling loads, and so forth.

In discussing the allocation of processing resources, we describe the allocation of raw processor time. Some objects in an agoric system might not purchase time this way, but might instead purchase interpretation services on a similar (or very different) basis. A system that can provide a market in raw processor time can serve as a foundation for more sophisticated services of this sort.

1.3. Rent-based storage management

Because storage needs will frequently be transient, we expect that rental from owners will emerge as a major means of holding storage. As with processor time, storage will have a price relative to other resources, and this price will vary across different media, locations, and times. As with processors, the long-term average price of storage in the system will reflect the external market price of adding more storage to the system, if owners are alert to opportunities for profit. We discuss allocation of raw storage space here, but a system that can provide a market in raw space can serve as a foundation for more sophisticated storage services.

The basic idea of our initial strategy and the emergent garbage collection algorithm is as follows:

- Landlords own storage space.
- They charge other objects rents at a rate determined through auction.
- Referenced objects (*consultants*) charge referencing objects (*clients*) retainer fees, using these to pay their rent (and the retainer fees charged by *their* consultants).
- When objects fail to pay rent they are evicted (that is, garbage collected).

These arrangements provide a natural way to free storage from unproductive uses. If an object cannot or will not pay its rent, then some other object must be bidding more for the available space (if no object was bidding for the space, its price would be zero). Since an object's income (and hence rent-paying ability) reflects the value placed on the object by its users, eviction of non-paying objects will typically improve the overall usefulness of the contents of storage.

Frequently-used consultants will be able to pay their rent out of their usage fees. Rarely-used (but referenced) consultants can charge their clients retainer fees adequate to cover their rent (and that of any consultants they themselves retain). In these relationships, pointers are bi-directional: a consultant also knows its clients. Unreferenced objects will be unable to earn usage fees or charge retainer fees; they will be unable to pay, and will be evicted, thereby accomplishing garbage collection (or forcing migration to a fixed-entry-price archive). This is the basis of the *market sweep* approach to garbage collection.

Rent-based storage management also allows a generalization of pointer types. Some systems distinguish between the traditional *strong pointers* and *weak pointers* [3]. A strong pointer retains a referenced object regardless of the cost: it represents an unbounded commitment to maintaining access. A weak pointer maintains access as long as the object has not been garbage collected, but does not itself cause the object to be retained. Weak pointers are an existing step toward economic storage management: they represent a small value placed on access—in effect, an infinitesimal value. This suggests a generalization in which an object will pay only a bounded amount for continued access to an object. This may be termed a *threshold pointer*. Thresholds may be set in various ways, for example, by limiting the total retainer fee that will be paid, or the total fee that will be paid in a given time period. When multiple threshold pointers reference an object, their strengths add; thus, they integrate information about the demand for retaining the object in a given region of storage. (As we will see, however, any situation in which a consultant asks retainer fees from multiple clients presents a challenge in incentive engineering—why should a client pay, if others will do so instead?)

Differing rents in differing media give objects an incentive to migrate to the most cost-effective locations. If clients offer a premium for fast service and demand service frequently, a consultant might best be located in RAM; if slow service is adequate and demand is low, a consultant might best be located on disk, or on a remote server. Caching decisions can thus be treated as a business location problem.

Rent-based storage management solves a standing problem of distributed garbage collection. Consider a loop of objects, each pointing to the next, each on a different company's machine, and all, collectively, garbage. Garbage collection could be accomplished by migrating objects to collapse the loop onto one machine, thus making its unreferenced nature locally visible [4]. But what if the companies don't fully trust one another? By sending the representation of an object to an untrusted machine, the algorithm would allow encapsulation to be violated, giving away access to critical objects and resources. With rent-based storage management, however, migration is unnecessary: since unreferenced loops have no net income but still must pay rent, they go broke and are evicted.

The problem of unreferenced loops crossing trust boundaries highlights the lack of a notion of payment-for-service in traditional approaches to storage management. Introducing this notion seems essential when hardware and data are separately owned. In its absence, distributed systems will be subject to problems in which one person (or entity) forces the retention of another's storage but has no incentive to free it.

As suggested earlier, we do not expect explicit rental arrangements to be economical for very small objects. The appropriate minimum scale is an open question; the ultimate test of answers to this question will be market success.

1.4. Design constraints

As we have noted, initial market strategies must satisfy various constraints, which fall into two classes. First, they must result in a programmable system; this can most easily be guaranteed by ensuring that they meet the familiar constraints that have evolved in systems programming practice. Second, they must result in a system with market incentives, making possible the evolution of the new programming practices expected in agoric open systems.

Systems programming constraints often guarantee some property regardless of cost—for example, guaranteeing that referenced objects will be retained. Sound market incentives require that all resources used be paid for, since to do otherwise in an evolving system would foster parasitism. These two constraints would seem to be in conflict. To resolve this, we introduce the notion of the *well-funded object.* An object is well-funded if it has immediate access to ample funds to pay for the computations it spawns. A well-funded object might typically represent a human user and fund some computations serving that user. These computations are required to satisfy traditional systems programming constraints only so long as the well-funded object remains solvent, that is, so long as the user is willing to pay their cost.

The chief systems-programming constraint in processor scheduling is that processes be scheduled—that is, that there be a way to arrange bidding such that a well-funded process can be guaranteed non-starvation and hence eventual execution. The chief market-compatibility constraints in processor scheduling are that processor prices fluctuate to adjust demand to the available supply, that objects be able to make scheduling decisions based on price information, and that opportunities for malicious strategies be limited—for example, that a process not be able to force a high price on a competing process while avoiding that high price itself.

Several systems-programming constraints are important in storage management. First, non-garbage—everything reachable by a chain of strong pointers leading from a well-funded object—must not be collected. Second, garbage—everything that is unreferenced and cannot induce other objects to reference or pay it—should eventually be collected. Finally, overhead costs should be reasonable: bookkeeping storage per object should be bounded and the computational burden of the algorithms should scale roughly linearly (at most) with the number of objects.

Market-compatibility constraints are also important in storage management. Objects should manage retainer-fee relationships such that there is an incentive for clients to pay retainers, lest there be an incentive to shirk. A consultant's total retainer fees (which amount to a price for its availability) should reflect real storage costs, to provide non-initial-strategy clients with a reasonable basis for decisions. Finally, objects should not require unbounded cash reserves to avoid improper garbage collection.

Non-initial-strategy objects need not themselves meet system-programming constraints, since they are free to act in any manner that furthers their market success. They will still typically require reasonable computational costs, smooth interaction with other strategies, and bounded cash reserves. A complex market-level object, however, will be unlikely to point strongly at an object having unpredictable or uncontrollable costs. It must therefore be prepared for such consultants to go away. It may also spawn low-priority processes; some of these may never run.

How can a complex object be prepared for loss of access? Current practice already provides many examples of objects able to deal with the unexpected unavailability of objects they use. Programs are frequently prepared for the breaking of inter-machine or inter-process connections, or for the inability to open an expected file. Files are commonly updated so that they are in a recoverable state even if they should suffer the sudden loss of the updater. Argus provides abortable transactions and exception handling [III]. Additional recovery mechanisms can be expected among complex objects in a market environment.

2. Processor scheduling

This section describes initial market strategies for both sellers and buyers. In processor scheduling, we will term the time-seller (or agent for the seller) an *auction house,* and a time-buyer (or agent for a buyer) a *bidder*. A system may have any number of competing sellers.

2.1. Auctioning processor time: the escalator algorithm

A standard approach to scheduling processes uses a "first-come, first-served" queue. A newly-ready process always joins the tail of the queue, and the processor always runs the process at the head of the queue. This ensures that each process will eventually run (regardless of processor demand), guaranteeing what is known as *non-starvation* or *fairness*. This mechanism does not enable market trade-offs among the needs of different processes, however. A natural approach to doing so is the "highest-bid, first-served" queue. This corresponds to auctioning time-slices, with the queue corresponding to an auction house. Naïvely applied, this would lead to disaster: if the market price of the processor stays above a process's posted bid, the process will never run, and hence never learn that it needs to raise its bid. This defines a central problem in auctioning processor time.

2.1.1. Auction-house initial strategies

A basic question in an auction-based strategy is the nature of the auction: kinds include the double auction, English auction, Dutch auction, and first-price and second-price sealed-bid auctions [5,6]. In a double auction, sellers offer lower and lower prices while buyers offer higher and higher prices until they meet. In the familiar English auction, buyers bid higher and higher prices until the process plateaus; the seller accepts the highest bid. In a Dutch auction, a seller offers lower and lower prices until a buyer claims the item at the present price. In a first-price sealed-bid auction, fixed bids are submitted, and the highest is accepted; in a second-price sealed-bid auction, the highest is accepted, but the highest bidder pays the amount bid by the second-highest.

These auction institutions have differing applicability to the sale of time slices. The double, English, and Dutch auctions (at least in naïve implementations) require that processes be active while bidding for the very processor they need in order to be active—a major problem. Sealed-bid auctions avoid this problem, but they fail to guarantee non-starvation: if the processor price remains above what a process has bid, it will never be scheduled—and if the process is never scheduled, it cannot raise its bid. Thus, auctioning processor time is a bit like trying to auction wakeup pills to a sleeping crowd.

The approach explored here will be a variant of a sealed-bid auction, but the choice between first- and second-price forms remains. In laboratory experiments with human bidders, second-price sealed-bid auctions are known to give results similar to those of English auctions, and both lead to efficient markets (as does the double auction) [5,6]. In the English auction, the winning bidder pays only slightly more than the second-highest bidder; a second-price sealed-bid auction yields a similar result directly. Dutch and first-price sealed-bid auctions lead to less efficient markets.

First-price sealed-bid auctions give an incentive to guess what the next-highest bid will be, and to bid just slightly more. This strategic guessing serves no useful purpose in a market system. Second-price auctions give an incentive to consider only the question: "At what given price would my best decision change from 'buy' to 'don't-buy'?" This is the price one should bid, since bidding any other price might result in buying (or not buying) when one should not. Estimating this price means estimating actual value, which serves a decidedly useful purpose in the market system.

We have selected a variant of a second-price, sealed-bid auction for our initial market strategy. It may be called an *escalating-bid auction.*

This system may be visualized as an auction house full of escalators (admittedly a strange image). A process enters the auction by placing a bid on one of the escalators—the greater the bid, the greater the initial height. Each escalator rises at a different rate, raising its bids at that rate. (A special stationary escalator holds fixed bids.) Together, the initial bid and escalation rate are a form of priority. A processor always runs the highest-bidding process. A house rule sets a maximum allowable initial bid—you can get on only at (say) the first five floors.

Each auction house owns or leases a processor, or a certain fraction of its operating time. Escalator data structures make the highest bid readily available (*i.e.*, each escalator is a priority queue). Each non-stationary escalator is characterized by a rate of escalation, escalationRate, measured in currency units per time unit. At a time t, the value of a bid of zero initial value placed on an escalator at time timeOfBid is simply escalationRate $\times$ (t - timeOfBid). A non-zero initial bid of value initialBid is assigned a virtual bid-time, timeOfBid, equal to t - (initialBid / escalationRate), and entered accordingly. Thus, each non-stationary escalator is marked with a fixed escalationRate and holds a current list of bids, sorted in timeOfBid order. Each bid includes its timeOfBid, a suspended process, and access to an expense account. The stationary escalator is a special case; instead of a timeOfBid it records a fixed initialBid. (A negative initialBid is acceptable on a moving escalator. We assume that two idle processes are entered with zero bids on the stationary escalator to avoid accepting a negative bid-value; the first always stands ready to run, at the price set by the second.)

To place a bid on an escalator, one sends a suspended process, an initial bid, and access to an expense account from which the auction house is to withdraw money. When the bid is placed, the auction house immediately withdraws from the expense account enough funds to cover the worst-case cost of handling that bid.

At the beginning of each time slice, the auction house examines the top bid on each escalator, taking the highest bid among them (and promoting its follower) while noting the second-highest bid (taking into account the newly-promoted bid). It then charges the high bidder the amount of the second-highest bid, and gives the high bidder a slice of processor time. If the highest bidder's expense account fails to cover the (escalated) bid, however, it is removed without running, and a bid equaling the balance of its expense account is entered for this bidder on the stationary escalator.

The escalating-bid auction seems well suited to the processor scheduling problem. It avoids the sleeping-bidder problem and it ensures that a processor can accept a bid at any time—crucial, when the commodity to be sold is as perishable as time itself.

2.1.2. Bidder initial strategies

A sufficient initial strategy for a bidder is simply to place a zero bid on the fastest escalator backed by the bidder's own expense account. Note that, among initial-strategy bidders, the escalator algorithm reduces to a round-robin scheduler. In a slight variation, a negativeinitialBid can be placed to ensure a delay of at least (-initialBid/escalationRate) until the bidder next runs.

2.1.3. Analysis

In an open system, where total processor capacity and demand will be responsive to market forces, the market price of time on a processor will be bounded. Accordingly, a bid placed on any non-stationary escalator will eventually grow large enough to ensure that it is accepted. Thus, non-starvation would be ensured.

Where too much of the demand is unresponsive to price, other conditions are necessary to ensure non-starvation, such as limiting the maximum initial bid, maxInitialBid, to some fixed value (the "fifth floor") as suggested above. Consider a process Z with a bid on a moving escalator. Z will either run or have its bid escalated past maxInitialBid within a fixed time; at that time, only a finite number of other processes can have bids higher than Z's, and if Z is riding the *fastest* escalator, no new process can be scheduled to run ahead of it. Thus, the auction house guarantees non-starvation to any process that follows the strategy of always

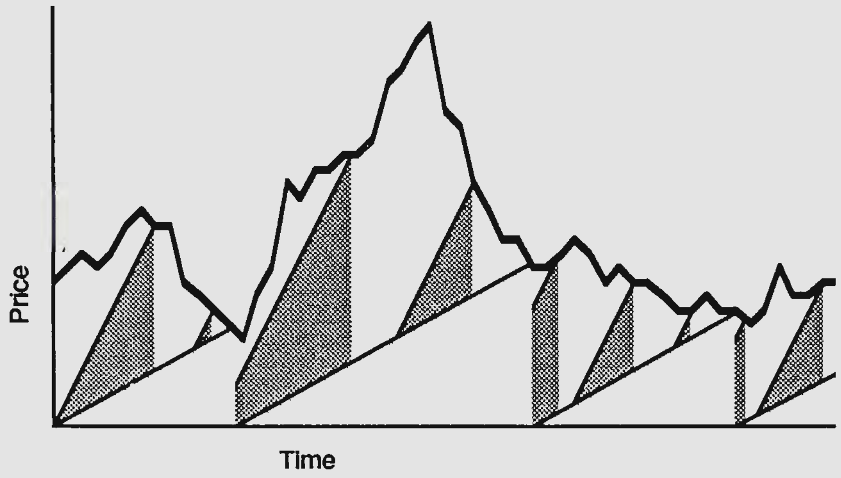


Figure 1: Escalating bids. *The jagged line above represents a hypothetical processor price history. The series of light triangles below represents the bid-history of a process that repeatedly reschedules itself with a zero initial bid; the dark triangles represent that of a similar process using a faster escalator. The processor price history reflects the bids of numerous other processes.*

entering a bid on the fastest escalator.

The relationship among bids, prices, and rates of use is simple in certain illustrative cases. Assume a stable price for time-slices, equal to P, and an escalator that raises bids at a rate R per time slice; an object that repeatedly reschedules itself after running by placing a zero initial bid on this escalator will receive a fraction of the processor time roughly equal to R/P. Consider an auction house in which a fixed number of processes repeatedly run and reschedule themselves, placing bids with zero initial values and a fixed distribution across the various escalators; assume further that bids are numerous enough and uniform enough to make second-highest bids approximately equal to highest bids. There will then be a steady-state price for a time-slice (with small fluctuations); this price will equal the sum over the escalators of the number of bids on each times the amount of escalation during a time-slice. (This quantity equals the per-time-slice increase in the sum of the bids, and all the money bid is eventually spent on processor time.) Non-zero initial bids will have an effect roughly like that of different escalation rates, and fluctuating rates of bid-placement will cause fluctuations in processor price.

Given fluctuating prices (see Figure 1), faster escalation rates will result in higher average costs per time slice. If scheduled at random times, rapidly-escalating bids will strike the market-price line at nearly random times (random sampling would hold strictly if escalation were infinitely fast). As may be seen, slowly escalating bids are unable to strike the price line at the top of sharp price peaks; they are more likely to strike the down-side of price troughs. Figure 1 also illustrates how a strategy of re-bidding at zero on an escalator after every run will, on the average, use more time-slices during broad troughs than during broad peaks, yielding a cost per time slice that is lower than the average cost; conversely, bids placed on fast escalators will pay a higher than average cost.

The overhead of the escalator algorithm is modest and insensitive to the number of bids being escalated. Assume N is the number of bids on an escalator and M is the number of escalators. Placing or removing a bid is then an operation taking a time proportional to log(N), given a suitable choice of escalator data structure (a priority queue). Finding the highest and second-highest bids by searching the top bids is an operation taking a time proportional to M.

2.1.4. Variations

The simplest auction-house initial strategy provides a fixed set of escalators, as described; more complex strategies could create and delete escalators to suit bidder demand. Other extensions would allow bidding for multiple time-slices as a block (up to some maximum size), or enable refunding payment on unused portions of a time slice (and starting the next full time-slice early). Where multiple processors are equally accessible, a single auction house could serve them all. Finally, the owner of a processor could run an auction procedure for a fraction of the available time slices and an entirely different procedure (perhaps some form of futures market for real-time scheduling) in another.

As described, the simplest bidder initial strategy is to schedule a zero bid on the fastest escalator. A more complex strategy might use a fast escalator only for fast service at a (likely) higher price, or slower escalators for slower service, at a (likely) lower price. A positive initial bid on a slow escalator can speed service while still giving better odds of running at a low price than does a bid on a fast escalator. Tasks of strictly limited value (which need not be completed) can be scheduled on the stationary escalator; they will run only if the price of processor time falls low enough. A regularly-scheduled rebidding agent can be used to implement a very broad class of strategies, taking into account new information from bid to bid.

There are several open issues in this approach to processor scheduling. These include finding procedures for:

- choosing maxInitialBid (where this parameter is needed),
- choosing the numbers and rates of escalators, and
- charging for bid-record storage.

In addition to solving the sleeping-bidder problem peculiar to process scheduling, the escalator algorithm provides a low-overhead auction procedure for allocating other resources that are naturally divided into time slices. For example, parameters for a bidding strategy could be part of a packet traversing a network, enabling the packets to bid for access to communication channels.

2.2. Expense accounts

We have described initial market strategies for the relationship between owners and bidders; we also need strategies among bidders, to ensure that they can pay for processing time. Since bidders typically need processing time in order to satisfy external requests, the initial market strategy should follow the dynamic structure of relationships created by request messages from client objects to their consultants.

When a client requests service from a consultant, we assume the client will pay to satisfy the request. We need an initial strategy that enables consultants to charge and clients to pay, all with a minimum of programmer attention (the following strategy does, however, require that objects distinguish between request and response messages). The *initial* strategy should itself provide neither profit nor loss, and hence should simply require that consultants charge for their operating costs, and that clients pay for them. This initial market strategy must (as always) interact smoothly with other strategies. The initial strategy must accommodate clients wishing to monitor charges or limit payments, and consultants wishing to charge less or more than their expenses (*e.g.*, to promote a new service or to collect royalties).

The initial strategy is as follows: Each process draws operating expenses from its *current expense account*. A client includes access to its current expense account in each outgoing request. The consultant then uses this account as its current expense account while satisfying the request. This strategy is identical to the protocol specified in the Act 2 language [7] for passing *sponsors*. Like expense accounts, these give bounded access to processor time [8].

In a set of objects following this initial market strategy, all computation serving an external request will be paid for by the account contained in that request. Since no computation will be cut off while that account remains solvent, well-funded computations will be completed.

In variations on this strategy, a consultant may charge according to whatever policy it wishes, since it is free to draw funds from the incoming account and to use a different account to pay for its computation. If a consultant requires a minimum sum to complete a computation, it can ensure access to this sum by transferring it to a new account at the outset.

A client may limit its payments to a consultant by sending a new account with the request and placing a limited sum in that account. This is like a threshold pointer, in that the client limits its liability at the risk of cutting off valid computation.

A client may monitor its payments to a consultant by sending a *shadow account* which passes charges through to an actual account while remembering their sum. When the consultant finishes, the client recovers the record of the total charges and shuts down the shadow account. This enables clients to accumulate cost information to guide further requests.

3. Storage management

In rent-based storage management, we again must specify strategies both for the relationships between buyers and sellers (here, renters and landlords) and for the relationships among renters (in their roles as clients and consultants). The latter are complex.

3.1. Renting memory: the rental-auction algorithm

In a fully competitive market for storage space, a landlord (having many competitors) will maximize revenue by seeking full storage utilization, setting its rental price at a level at which supply equals demand (the *market-clearing rate)*. An auction-based initial market strategy can approximate this rather well.

A landlord maintains (or uses) an auction house which keeps two data structures, a *bid list* and a *drop list*. The bid list records requests for blocks of storage; each request is associated with an object, a desired quantity of storage (limited to a maximum request, maxBlockRequest, of perhaps 1% of total local storage), and a price—per unit of memory, per of unit of time—bid for acquiring it (bidPrice). Bids are accompanied by deposits to cover handling charges. The drop list records already-leased blocks of storage, each associated with an object, a block size, and a unit rental price at which the object would prefer to release it (dropPrice). The lists are ordered by bidPrice and dropPrice respectively. A running total is kept of the amount of wasted space.

We consider the bid list to also contain an infinite number of bids at zero rental price for atomic blocks of storage to be allocated to a *free memory sponge* object. The sponge will be allocated memory only when no one has any use for free storage; any memory so allocated is entered on the drop list with a zero price. In a mature agoric open system, the demand for memory space should be enormous at low enough prices. With a charge-per-use policy, there is no bound to the amount of software that would migrate to a machine offering a zero storage price; storage of debugging traces and caching of calculated results would likewise expand at a zero storage price. Therefore, one would not expect to see a zero price or see any memory allocated to the sponge.

Fresh unheld space becomes available at the beginning of operations, when space is vacated, when objects are evicted for nonpayment of rent, or when more memory is purchased and added to the system. The auction house then accepts bids from the top of the bid-list, highest bidPrice first. It continues allocating blocks to bidders until it encounters a bid for a block larger than the remaining unheld space. This bid is shelved, the allocation process stops, and the price of this unsuccessful bid is taken as the rental price of storage for all objects during the next time segment.

If, as expected, the blocks requested total more than the storage available, then the maximum unallocated storage will be smaller than maxBlockRequest. If this is 1% of the total, storage utilization will be at least 99%. For example, consider a computer with ten megabytes of main memory and a memory management unit that maps addresses in 1kilobyte blocks. Memory to be allocated and traded would consist of integral numbers of these 1kilobyte blocks (which can be mapped arbitrarily, hence we can ignore fragmentation). One percent of 10 megabytes is 100 kilobytes, so this is maxBlockRequest and the largest amount that can be wasted by the above procedure. We assume that any object needing a block bigger than 100 kilobytes can afford the trouble of acquiring it 100 kilobytes at a time.

When a new bid is placed, its bidPrice is compared to the highest bidPrice on the bid list. If it is lower, it is placed on the bid list; if it is equal to or greater than the highest bidPrice and equal to or less than the lowest dropPrice on the drop list, then it is accepted if it requests a block that can be allocated from unheld space, and otherwise is placed on the bid list. If it is greater than the lowest dropPrice, then room may be freed for it.

In this case, the auction house attempts to identify enough space to accommodate the new bidder, starting with the unheld storage and then proceeding to the held blocks lowest on the drop list. Objects responsible for identified blocks are asked to vacate them or to set a higher dropPrice. On vacating, renters are refunded the unused portion of their rent money. This process stops when (1) enough space has been freed, or (2) a block is encountered having a drop price equal to or greater than the bidPrice of the new bidder. In case (1), the new bidder receives storage space and is placed on the drop list at a dropPrice of its choosing; in case (2), the new bidder is placed at the head of the bid list. In either case, the rental price of storage becomes the bidPrice of the highest unsuccessful bidder.

To guarantee that resources used will be paid for (and avoid incentives for the evolution of parasitic software), landlords must require payment for a rent period in advance. This payment should cover the cost of the next billing cycle and include a deposit to cover the cost of deallocating memory and of any special services specified in the lease agreement, such as erasure of vacated space (a sort of cleaning deposit). Rental rates will fluctuate during a rent period, with the length of a rent period varying as the inverse of the average rental rate.

Landlords can accept lease agreements of varying lengths, requiring varying amounts of pre-paid rent to allow objects to tune their storage management overhead. They can likewise agree to provide a period of advance notice before collecting rent, giving the renter time to raise money, find alternative storage, or close out its affairs.

A more complex strategy would offer prompt storage allocation (from pre-emptable cache or unheld storage), charging a premium for this service. Alternatively, this and other services could be provided by renters subletting space in their blocks. A useful service would allow a renter to split off a piece of its storage block and post a new drop-list entry for it, allowing the sale of portions of allocated blocks without the overhead of the auction house procedure.

3.2. The market-sweep algorithms

An initial market strategy for renters is to get space by placing high (perhaps escalating) bids, and to keep it by paying whatever is necessary, so long as funds hold out. The challenge is to have funds hold out while the renter should stay, and eventually run out when the renter should vacate. Since a consultant must pay its rent in order to serve referencing clients, the initial market strategy follows the referencing structure among consultants and their clients. This structure is a directed graph containing cycles and changing over time. As a result, these strategies are more complex than those above.

A consultant must not be evicted if can be reached through a chain of strong pointers starting from a well-funded object (*i.e.*, non-garbage must not be collected). Many objects will pay their rent out of fees charged for their services, but some objects—though never before used—may be of great value in rare contingencies: consider an object that contains plans for coping with the next terrestrial asteroid impact. Objects that are needed but rarely used must survive by charging their clients retainer fees; an initial strategy must assume this worst case.

A system based on retainer fees must avoid several problems. In one approach, objects would, when charged rent, send *alert messages* to their clients, asking for the needed sum; these clients would do likewise until a solvent object was found to pay the bill. This system has low capital costs, requiring little cash on hand, but it leads to an explosion of circulating alert messages, and hence to unacceptable transaction costs. In an alternative approach, objects would keep enough cash on hand to cover worst-case rent and retainer fees. This system has minimal transaction costs per rent period, but in the absence of information on worst-case rents and fees, capital requirements are unbounded and garbage collection would be indefinitely postponed.

For a system to be acceptable, transaction and capital costs must both be bounded. Transactions should require on the order of one message per pointer per rent cycle, capital required should be some fixed multiple of per-cycle expenses, and the per-cycle retainer fees paid by a client should be reasonably related to the rent paid by its dependent consultants.

Satisfying these constraints proves to be difficult. The component algorithms of the market-sweep approach include the following:

• The *dividend algorithm,* an initial market strategy which provides incentives to pay retainer fees to an object and provides it with estimates of future use. It has strategic properties which should make it useful in contexts outside storage management.

• The *retainer-fee algorithm,* an elementary algorithm for billing clients. This provides for cash flow in normal circumstances.

• The *alert algorithm,* which provides for fast cash flow when needed to prevent improper eviction. It is an initial strategy aimed at guaranteeing systems programming constraints, but it involves a race which can fail given a sufficiently unfavorable combination of cash reserves, rental notice period, message-passing speed, and message path length.

• The *base-demand algorithm,* which provides estimates of future cash requirements to minimize the need for alerts. It is essentially heuristic, aimed at tuning reserves and estimating costs. As a cost-estimater, it can underestimate the drop in retainer charges that will occur when a new client points into a cyclic reference structure.

3.2.1. The dividend algorithm

Raising money by simply dividing the total retainer charge equally among several directly-pointing clients won't work; it suffers from the classic public-goods problem: each client has an incentive to shirk, as long as another will pay. To pay to retain a consultant while a competitor uses it without paying is an evolutionarily unstable strategy—clients following it will lose in price competition. Multiple clients can pretend to be single clients (and thus cut their liability) by pointing via a middleman (Figure 2). To make simple retainer-charging schemes of this sort work would require peculiar and uneconomical limitations on object interactions, such as (somehow) preventing one object from serving as a middleman for another, or preventing a consultant from offering service to new clients.

A better alternative is for the consultant to collect a large enough surcharge *per use* to compensate all clients (or their heirs and assigns) for their earlier retainer payments. This approach converts retainer-payment into a form of investment, to be repaid through dividends raised by imposing a per-use surcharge. To make such investment attractive, dividends must be propor-

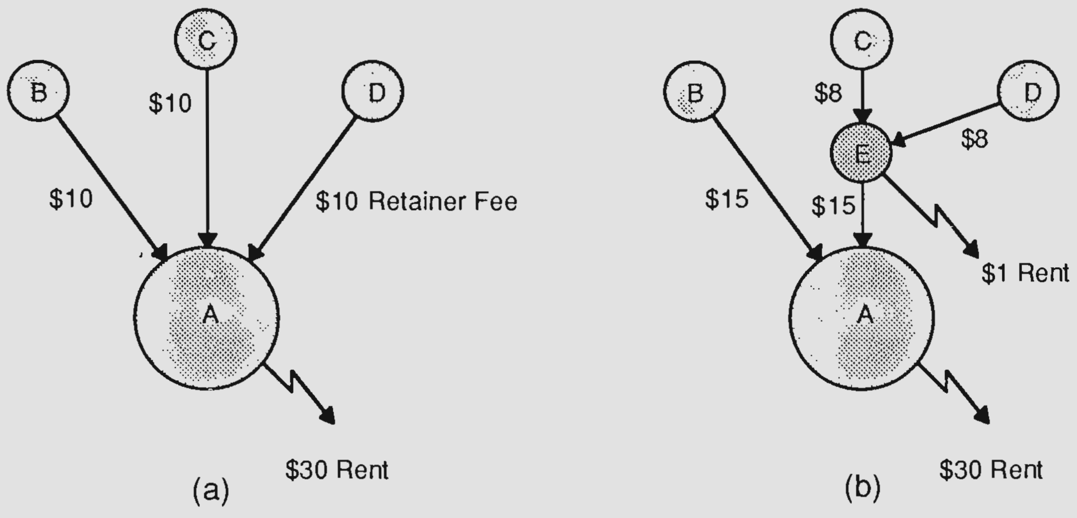


Figure 2: Client conspiracy. *Straight arrows represent clients pointing to consultants; they are labeled with the corresponding retainer fee payments; crooked arrows represent rent payments.* ***(a)*** *shows the charges if a consultant's liabilities are split equally among its clients;* ***(b)*** *shows how C and D can reduce their payments by pointing through a forwarder, E.*

tional to the amounts invested, and must include compensation for the risk that a consultant may in reality be used seldom (or not at all), yielding few or no dividends.

This raises the problem of estimating a consultant's future use rate (or use probability). The lower the expected use rate, the higher the per-use charges and dividends must be to compensate clients for their investment. These use-rate estimates must somehow reflect the client's own judgment, lest clients be unwilling or too-willing to pay.

Future use rates for a consultant could be estimated using the sum, average, or median of future use rates estimated and reported by its clients. But why should these reports be accurate? Mechanisms which yield estimates proportional to the sum or average are clearly unstable: if all clients share equally in retainer payments, high-use-rate clients should estimate an infinite use rate, to drive the per-use surcharge for dividend payments to zero; low-use-rate clients should estimate a zero rate, to maximize their dividends. Use of the median mechanism throws away information (*e.g.*, a single high-use-rate client among several low-use-rate clients will have no effect on the estimate); this presumably leads to wasteful strategic behavior.

Another approach would be to accept bids for the privilege of investing, with the winning bidder asking the smallest surcharge-and-dividend per future use. This approach is stable, since payment is voluntary and will typically be justified at some level of dividend, but it fails to integrate market information effectively. Instead, it encourages clients to give a falsely-high impression of their future use rates, to drive down others' bids and hence their own future usage charges. Thus, the prospective bidder must guess others' actions based on what may be actual disinformation. Further, the special role of the low bidder encourages messy strategic behavior.

One would like an algorithm for collecting retainer payments and paying dividends that has better properties. Ideally, it should give clients an incentive to report accurate use estimates, enabling a synthesis of estimates made by those in the best position to know; further, it should provide incentives for simple strategic behavior in typical situations, and it should be insensitive to issues of entity definition—to whether one treats a buying club as an object or as a collection of objects.

3.2.1.1. Description

This section provides a brief, abstract description of the dividend algorithm in its simplest form. Later sections describe its operation more informally, analyze its properties, and describe a slight modification that yields a more practical version.

Definitions of variables

R = a retainer-seeking object (the consultant)

t = an index specifying a time at which R collects retainer fees

F_t = the total fee required by R at time t

C_j = a client of R

m = the number of clients referencing object R

N_{jt} = the reported estimate of number of future uses of R by C_j

$S_{jt} = F_t N_{jt} \big/ \sum_{i=1}^{m} N_{it}$ = the share of F_t requested of C_j

P_{jt} = the amount actually paid by C_j (is $\leq S_{jt}$)

$N'_{jt} = N_{jt} P_{jt}/S_{jt}$ = the "effective" reported estimate of future uses

F'_t = the total amount actually collected by R at t, $= \sum_{i=1}^{m} P_{it}$

D_{jt} = the "dividend sum" at t for C_j

$A_{jt} = P_{jt} \big/ \sum_{i=1}^{m} N'_{it} = F'_t N'_{jt} \big/ \left(\sum_{i=1}^{m} N'_{it}\right)^2$ = the amount to be added to the dividend sum

n_{jt} = the actual number of future uses by C_j, counting from time t

E_{jt} = the expected net cost of future uses (resulting from actions *at time t)*

L_t = time since R last collected retainer fees

W = time weighting factor (for the time-weighted version of the dividend algorithm)

The dividend algorithm proceeds as follows: at time t, object R has clients C_i (i = 1 to m) and seeks a total retainer fee F_t. For each client C_j, R maintains a dividend sum D_{jt}, created and initialized as zero when C_j first pointed at R. At time t, R asks each client C_j for a number N_{jt}. R then asks each C_j for an amount of money

$$S_{jt} = F_t N_{jt} \Big/ \sum_{i=1}^{m} N_{it}.$$

From C_j, R receives (after using charging algorithms described in the next section) an amount P_{jt}. Then, R replaces each N_{jt} (as needed) with $N'_{jt} = N_{jt}P_{jt}/S_{jt}$, and (in the unweighted form of the dividend algorithm) sets

$$D_{jt} = D_{j(t-1)} + A_{jt}, \text{ where } A_{jt} = P_{jt}\Big/\sum_{i=1}^{m} N'_{it}.$$

When R is used between times t and t+1 (by any object, whether among C_i or not), R collects a surcharge equal to

$$\sum_{i=1}^{m} D_{it},$$

(plus an amount to cover actual service costs, profits, and so forth—these charges are ignored in the following, since they have no effect on rent and retainer strategies). R then pays a dividend equal to D_{it} to each client C_i.

3.2.1.2. Basic analysis

Clients evolved under competitive pressures will tend to act so as to minimize the expected net costs of their actions. These costs may be analyzed as follows.

Assume that client C_j first paid a retainer fee to R at time $t = T_1$. For a client C_j, total expected costs and paybacks are a function of F_t, N_{it}, and P_{it} (for $i = 1$ to m, and $t = T_1$ to ∞). The analysis of expected costs may be simplified by noting how payments are mediated though the dividend sum D_j. Let us represent a sum over $i = 1$ to m, $i \neq j$ as

$$\sum_{i \neq j} X_i.$$

The cost to C_j of using R at a time T_2 is

$$\sum_{i \neq j} D_{it} = \sum_{t=T_1}^{T_2} \sum_{i \neq j} A_{it}.$$

Payback at the time of use by another (at time T_2) is

$$D_{jt} = \sum_{t=T_1}^{T_2} A_{jt}.$$

Since these costs and paybacks are a simple sum of contributions from different times, distinct contributions can be identified from each time. Thus one can isolate the consequences to C_j that result from retainer-payment actions (by C_j and others) at any given time: these consequences are independent of retainer-payment actions at other times. This simplifies the analysis of optimal strategies. The net cost to C_j *resulting from actions at time t* (represented as E_{jt}) is simply (1) the immediate cost of paying the retainer fee, plus (2) the net surcharge per C_j's own use (resulting from the other clients' increments to the dividend sum at time t multiplied by the future number of uses by C_j), minus (3) the dividends from each of the other clients' uses (resulting from C_j's own increment to the dividend sum at time t multiplied by the total future number of uses by others). This is

$$E_{jt} = P_{jt} + \Big(\sum_{i \neq j} A_{it}\Big) n_{jt} - A_{jt}\Big(\sum_{i \neq j} n_{it}\Big),$$

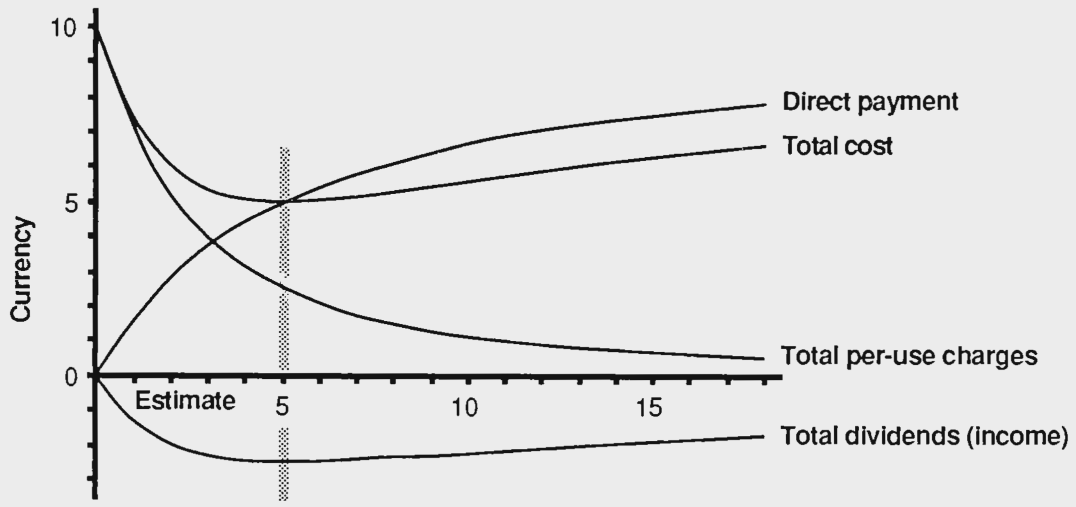


Figure 3: Dividends and incentives. *Assume that client* C_j *will make five future uses of object* R, *while its other clients together will make a total of five uses. Differing usage estimates* N'_{jt} *will then incur long-run costs shown above, assuming that* R *charges a retainer fee of ten currency units, and that the other clients' estimates sum to five. Note that* C_j*'s total cost is minimized by reporting a correct estimate of its own future use.*

$$= F'_t \left(\sum_{i=1}^{m} N'_{it}\right)^{-1} \left[n_{jt} + N'_{jt}\left(1 - \sum_{i=1}^{m} n_{it} \Big/ \sum_{i=1}^{m} N'_{it}\right)\right].$$

Considered as a function of n_{jt}, this has a minimum at

$$N'_{jt\,(\text{optimal})} = \frac{2\, n_{jt} + \left(\sum_{i\neq j} n_{it} - \sum_{i\neq j} N'_{it}\right)}{1 + \left(\sum_{i\neq j} n_{it} \Big/ \sum_{i\neq j} N'_{it}\right)}.$$

3.2.1.3. Analysis of incentives

The above equation implies that if

$$\sum_{i\neq j} N'_{it} = \sum_{i\neq j} n_{it},$$

then the optimum value of N'_{jt} for client C_j to submit equals n_{jt}. That is, if other clients are making accurate predictions of their future usage, then one can minimize costs by accurately predicting one's own usage. Cost allocation is uniform, with desirable incentives. After removal of a fixed sum to cover R's expenses, all charges are redistributed among the clients in a zero-sum game. If all clients accurately estimate their own usage, then each client's net charges are strictly proportional to its usage. This provides a level playing field for large and small entities, avoiding perverse incentives. Since $N'_{jt\,(\text{optimal})} = n_{jt}$, and since n_{jt} represents real uses, pooling or splitting demand among objects can make no strategic difference.

If other clients suffer from a systematic bias in their usage estimates, this benefits those that estimate their own use correctly. All the inaccurate clients can be modeled as one large, inaccurate client which (by hypothesis) is in an environment in which the remaining client(s) make correct estimates. Accordingly, its inaccuracy is suboptimal, causing losses. As this is a zero-sum game, those losses accrue to the accurate estimators.

But if a client knows that other clients are systematically submitting inaccurate estimates of their future usage, *and* knows the direction of their bias, it can bias its own estimate to improve its expected earnings. The direction of optimal bias—whether in the same or opposite direction to the others' bias—depends on additional knowledge of the relative magnitudes of the usage rates involved. It should be rare for a client to have all this knowledge. In a typical round, a client submits a number, then pays a request. It has no direct way to infer others' estimates or their actual future usage rates.

If others' total estimates are known to mistakenly equal zero, C_j's formally optimal value of N'_{jt} is zero, driving the future charge per use to infinity, and giving infinite expected dividend revenue to C_j (assuming demand for R is truly independent of its surcharge!). However, consideration of the real relation between price and demand will give different results, in which R is in price competition both with R's competitors and with any alternative copies of R, and in which the client-investor C_j must be viewed as a co-provider of R's service. Further, if R (or the creator of R) expects future uses from yet-unaccounted-for clients, then self-investment (by acting as a virtual client) makes economic sense. This would raise the effective total of others' estimates above zero.

In addition to the price-sensitivity of demand, the fixed transaction costs of paying retainer fees modify these conclusions somewhat, giving low-rate users an incentive not to participate in the process. If enough do not, the resulting underestimate of usage will give a positive return on investment to the clients that do, covering their transaction costs. A full analysis of optimal behavior seems likely to be complex.

3.2.1.4. Adding a time horizon

A major problem remains with this form of the algorithm: the magnitude of the dividend accounts grows steadily over time, without bound. There is no equilibrium cost per use, given finite estimates N'_{jt} by the clients, even given provision of an identical service to identical clients at a uniform rate. If clients expect an unbounded number of future uses, the algorithm becomes indeterminate regarding allocation of retainer shares to clients, refunds of invested sums become infinitesimal, and full payback of invested sums is indefinitely postponed. It therefore makes sense to investigate a broader family of dividend algorithms.

Consider a consultant R that (according to contract) will be in existence only until a time T. The above algorithm may be applied, and retains all its properties, given that the clients notice that their uses will (of necessity) cease at time T. This would continue to hold if R were immediately replaced with a new instance of itself having all dividend accounts initialized to zero. Clients could continue receiving service, but their optimal values of N'_{jt} before T would take account only of expected uses before T. The same analysis would hold if R simply zeroed its accounts at T, again according to contract.

In general, a consultant R can announce a policy in which all dividend accounts are to be multiplied by a time-dependent factor $w(t)$; in the above case, this factor is one before T and zero afterwards. Since expected costs and resulting strategic decisions are dependent only on expectations, actions, and policies in the current time period, a different weighting function could be announced for each period.

Assume that each client C_j will use R (counting from the present time t) at times τ_{j1}, τ_{j2}, τ_{j3} .... Represent the weighting function applied to the dividend sums (relative to the present time period t) as $w_t(\tau)$. The net cost to C_j resulting from actions at time t is then

$$E_{jt} = P_{jt} + \Big(\sum_{i\neq j} A_{it}\Big) \sum_{k=1}^{\infty} w_t(\tau_{jk}) - A_{jt}\Big(\sum_{i\neq j} \sum_{k=1}^{\infty} w_t(\tau_{ik})\Big)$$

But this is simply the original expression for E_{jt} with the substitution

$$n_{jt} = \sum_{k=1}^{\infty} w_t(\tau_{jk}).$$

Hence the analysis proceeds as before, but with sums of time-weighted uses replacing sums of uses; the previous analysis becomes a special case in which all weights are unity.

This immediately makes available a family of strategies sharing desirable properties. A simple member of the family is $w(t) = \exp(-Wt)$, where W is non-negative. This may be implemented in approximate form to create an initial-strategy dividend algorithm in which, in each round, R sets

$$D_{jt} = D_{j(t-1)} \exp(-W L_t) + A_{jt}.$$

For a uniform rate of use equal to U_j (the time-average of dn_{jt}/dt), the optimal value of N_{jt} is approximately U_j/W; this is exact, in the limit of short time periods L_t. As we shall see, W should *not* be set based on time-value-of-money considerations. Though the function is exponential, it does not represent compound interest.

This algorithm retains the stability and incentive properties of the first algorithm described. In addition, it yields a stable cost per use, given stable total retainer fees and usage-rates. Charges per use are still equal for all users, if all users estimate their usage correctly. Further, this algorithm asks clients for estimates of usage in an (effectively) bounded time interval—that is, it asks them for an estimate they may plausibly be able to make.

The parameter W can be heuristically tuned subject to the constraint that it be non-negative. In general, it should perhaps be tuned to make

$$\sum_{i=1}^{m} N'_{it} \approx 1.$$

This would change the sense of what is being estimated from the number of future uses to the probability of use within a bounded time interval. An object R may at any time announce that future estimates will be entered into new dividend accounts subject to a new function $w(t)$, so long as it pays dividends that result from summing the results of the new accounts with the results of the old accounts (which must be updated according to the old algorithm). This maintains all the incentive properties described above and allows retuning of W in a fair way, at the expense of additional overhead.

This algorithm allows a natural way to account for the time value of money, which may be important, since objects recover their investments only after a delay. If all clients submit low estimates, then all will receive greater dividends when R is used; this corresponds to receiving a return on their investment. For example, if they all bias their estimates by assuming a slightly greater value of W than R uses in its calculations, then the result will be as if they receive a certain rate of interest while their investments are repaid. (This holds on the average, assuming that actual use rates match expected use rates.) If different objects seek different rates of return, strategic considerations become more complex. Time-value-of-money considerations should be small in systems open to the external market, because market interest rates measured in percent per year are tiny per day or second. Long-run interest rates will equilibrate in a connected open system—investment will move toward higher rates, driving them down.

3.2.1.5. Accounting costs

A remaining problem with this algorithm is that R must maintain a dividend account for every client ever charged a retainer. This may be corrected in a way that demonstrates a generally-applicable principle for lowering accounting overhead.

What is important to the incentive structure of an algorithm (in the absence of risk-averseness considerations, as is appropriate with small enough sums of money) is not that *actual costs* have a certain magnitude, but that *average costs* do so. Random variations in actual expenses and payments make no difference if the amounts are small and the averages are correct. Accordingly, with proper attention to these points and to conservation of currency, charges and payments may be rounded or made on a statistical basis.

In the present case, we seek a principled way to cut off payments to former clients, cutting short the long, exponential tail of the dividend account. This can be done by freezing the magnitude of the account when it reaches a small-enough level, and then giving the account a suitable half-life for total deletion (using decisions based on random numbers to give a certain probability of deletion per unit time). This leaves the expected payback in all future periods unchanged, but makes the expected cost of maintaining the account asymptotically approach zero.

3.2.1.6. Circulation of usage estimates

Leaving aside the small correction for the time value of money, an estimate N'_{jt} may be interpreted as indicating a rate of use equal to $W\,N'_{jt}$. The quantity

$$W \sum_{i=1}^{m} N'_{it}$$

is then R's total expected rate of use, which R can use in estimating the rate at which it will use its own consultants, thereby propagating usage information through the system. The dividend algorithm thus provides local incentives for the combination and propagation of accurate estimates of future service demand, perhaps making possible sophisticated heuristics for anticipatory resource allocation—heuristics that reflect global conditions through purely local interactions. A conservative initial market strategy, however, might be to base usage estimates initially on some global average of initial-object-usage rates, and later on actual experience.

3.2.1.7. Open problems: the dividend algorithm

Several open problems are associated with the dividend algorithm. These include selecting an appropriate value of W and choosing an initial usage estimate in the absence of prior history.

A particularly interesting problem is the exploration of strategies which rapidly propagate future-usage estimates though a network of objects. If R's clients report increases in their expected usage, then R very likely has good reason to report increases in its expected usage of its consultants. General rules for revising these estimates must be stable in the presence of cyclic reference structures, and stable in the presence of clever, self-interested participants.

3.2.2. Normal money flow: the retainer-fee algorithm

The retainer-fee algorithm is the basic strategy for collecting funds to cover an object's rent and retainer-fee obligations. We earlier described initial market strategies as a sort of scaffolding for building a market. We expect the dividend algorithm just described to be scaffolding of a sort that eventually becomes a structural member; the retainer-fee and other initial-strategy algorithms for storage management seem like scaffolding of a sort one expects to be replaced as the construction proceeds. They raise fewer issues of incentives and strategic stability and are intended chiefly to ensure adequate money flow on a heuristic basis.

The retainer-fee algorithm proceeds as follows. In a given cycle each renter-object R has a number of clients, numberOfClients, and a current balance, currentBalance; it calculates (as discussed below) a desiredBalance (a target cash reserve) and a balanceReserve to be set aside for certain classes of payment. The latter is chosen such that the expected expenses for the next rent cycle can be paid without dipping into the reserve. Section 3.2.4, on the base demand algorithm, will explain how these expenses are estimated.

When asked to pay rent, R first calculates a total retainer fee F_t by calculating the amount (if any) by which its currentBalance falls short of its desiredBalance. R then charges each of its clients a retainer fee S_{jt} calculated through the dividend algorithm. (To this is added a surcharge, not counted in the dividend algorithm, to cover the billing cost; from an incentive perspective, this surcharge simply adds to the transaction costs discussed in the section on the dividend algorithm.) Clients pay these charges from their available funds, if they can. The dividend algorithm provides incentives to pay, since shortfalls will make $N'_{jt} < N_{jt}$.

If all clients pay in full, then R achieves a balance equaling its desiredBalance and is finished. If any fail to pay in full within a fixed time limit, those clients that paid their shares are asked for the remaining sum by iterating the retainer fee and dividend algorithms on these clients, using the values of N_{jt} they reported in the first round, and defining F_t as the remaining sum to be collected. This maintains the incentive structure of the dividend algorithm.

Since this process always eliminates a client from consideration, it can be iterated (counting the first round) no more than numberOfClients times. Given the addition of a billing charge, the maximum un-reimbursed message cost at any time is numberOfClients times the cost per message, hence the cash on hand needed to follow this protocol is strictly bounded and may be covered by a fixed per-client deposit.

If iteration of this request process fails to produce enough money to prevent an improper eviction, further measures must be taken. These are the subject of the alert algorithm.

3.2.2.1. Open problems: the retainer fee algorithm

Two open problems related to the retainer-fee algorithm involve essentially heuristic choices of parameters. One is the choice of how much cash to keep on hand to meet cash-flow contingencies, another is the choice of the length of a pre-paid rent period. An initial market strategy for the former is presented below as the base-demand algorithm. The latter depends on (at least) the cost of storage rental, the cost of processing rent requests, and the likelihood (as a function of time) that one should vacate storage.

Generation scavenging [9] and the Lieberman-Hewitt garbage collection algorithm [1] both rely on the insight that objects have a high infant mortality—that a good predictor of the longevity of an object is its age. Both check new objects more frequently, thereby collecting more garbage with less effort and cost. Our initial strategy can do likewise simply by prepaying rent for longer times as the renter ages.

3.2.3. Special money flow: the alert algorithm

If the retainer-fee algorithm fails to produce enough money to prevent R's eviction, R sends each of its clients an alert message. Strong pointing fundamentally means responding to alert messages appropriately: other money-handling procedures can fail, but (with caveats about execution time, notice, and rent periods) an unbroken chain of correct alert-processing objects leading back to a well-funded object will suffice to retain R in storage.

The idea of the alert process is to send requests for funds as far up and out through the chain of client relationships as is needed to find an object able to supply ample money. Failure to collect the needed money is assumed to imply that no solvent entity is willing to pay for continued storage of the renter, which may therefore be garbage-collected.

An algorithm for accomplishing this is described in Figures 4 and 5, and code implementing it is listed in the appendix to this paper. Recipients of alert messages first seek funds through application of the retainer-fee algorithm, then send further alert messages if needed. Propagation of alert messages in endless loops is avoided by giving each a unique identifier; objects refuse all but the first alert message with a given identifier.

All alert messages seek the full funds needed to satisfy the original alerter, plus enough to compensate the participating objects for their handling costs. This enables them to maintain a balance (the alertReserve) adequate to process further alert messages. Maintaining an alertReserve is like keeping a quarter for use in emergencies. Should you ever need to use it, you should use that phone call not only to take care of the emergency, but also to request a new quarter.

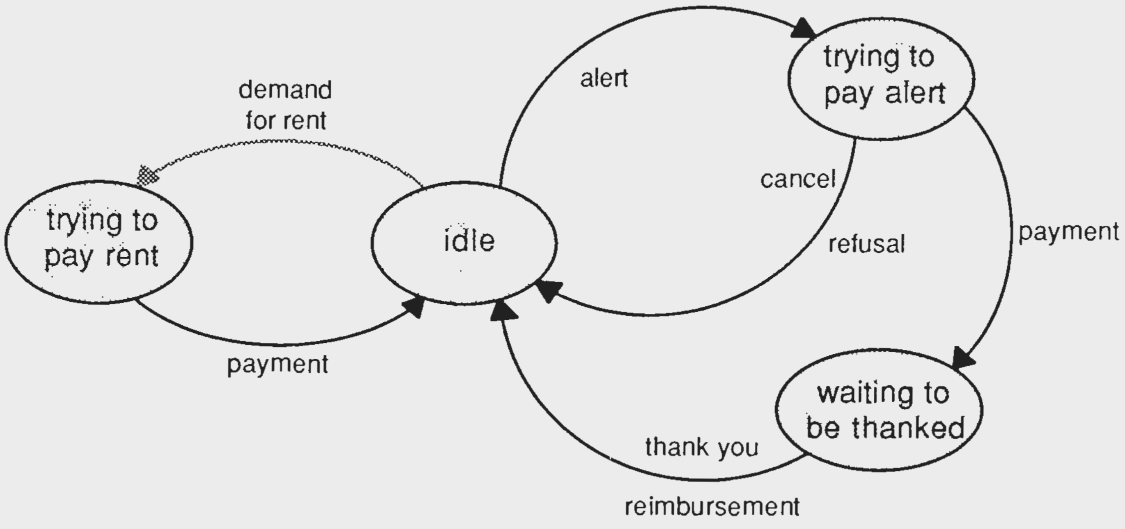


Figure 4: State transitions of the alert algorithm. *Ellipses represent an object's states; labels on arrows show which messages may cause it to change to another state. All states and messages are shown in Figure 5 except the demand-for-rent message (here drawn in grey).*

When a client pays the requested amount, the recipient sends *cancel* messages to its other clients, informing them that payment is no longer necessary. Due to asynchrony, however, multiple clients may pay before being cancelled. Further, a payment propagating down a chain of client-consultant references may be met by a corresponding cancellation propagating up the same chain. When a consultant finds it has received an unnecessary payment, it reimburses the payer-client. If the payer itself had merely passed the payment down, then it needs to pass the reimbursement up to *its* payer-client; thus, it needs to remember which client paid it. Since we require bounded storage costs for the algorithm, the payer needs to know when it can forget this knowledge. This occurs when the payer is reimbursed or thanked for payment—when an alert payment is actually used for rent, thank you messages propagate back up the path the payment came from. This algorithm allows a client to process only a single alert message at a time (another source of potential delay). A client must have enough storage to queue up one message per consultant, and enough cash on hand to send one message per client, hence the resources required to follow this protocol are strictly bounded.

From the perspective of the dividend algorithm, the alert process may be viewed as another iteration of the retainer-fee request cycle. Accordingly, objects that are prepared to pay alert-message requests promptly will have a competitive advantage over those that are not.

3.2.3.1. Open problems: the alert algorithm

The greatest problem with this alert-message mechanism is the time it requires. Even in the parallel-processing case, the time needed is proportional to the distance from a distressed to a solvent object. In the sense of guaranteeing non-collection of non-garbage objects, it works in the worst case only under the idealized assumption that alert processing times are negligible in

States / Messages	**Trying to pay rent** (alertID, alertQ, count)	**Trying to pay alert** (alertID, alertQ, count, alerter)	**Waiting to be thanked** (alertID, alertQ, payer)	**Idle**
Alert (alertID, alerter, amount)	Send refusal to alerter			If can pay amount, pay it; else alert clients, become "trying to pay alert"
If alert IDs differ:	If can pay amount, pay it; else place on alertQ			
Cancel (alertID)	Ignore cancel (payment is coming)	Cancel clients; process next alert, or idle	Ignore cancel (reimbursement is coming)	Ignore cancel
If alert IDs differ:	If for alert on alertQ, dequeue and ignore alert; else ignore cancel			
Thank you (alertID)	Error	Error	Thank payer; process next alert or idle	Ignore thank you
If alert IDs differ:	Ignore thank you			
Reimbursement (alertID, check)	Error	Error	Reimburse payer; process next alert or idle	Accept reimbursement
If alert IDs differ:	Accept reimbursement			
Payment (alertID, check, payer)	Pay rent, thank payer, cancel other clients, process next alert or idle	Pay alert, cancel other clients, wait to be thanked	Reimburse payer	Reimburse payer
If alert IDs differ:	Reimburse payer			
Refusal (alertID)	If all clients have refused, liquidate self	If all clients have refused, refuse alert, process next alert or idle	Ignore refusal	Ignore refusal
If alert IDs differ:	Ignore refusal			

Figure 5: Alert algorithm states and messages. *Columns represent possible object states (with state variables); rows represent messages (with arguments). Rows are split according to whether the* alertID *in a received message matches that remembered in the state variable. If so, the message concerns the same alert. The first four messages propagate from consultants to clients; the last two, from clients to consultants.*

comparison to rent pay periods. In practice, this algorithm's range of effectiveness will depend on the real times involved. The outstanding open problem is to develop an algorithm which avoids these delay problems in ensuring that non-garbage objects receive the money they need, or to develop clear (preferably locally-computable) bounds on the correctness of the alert algorithm—that is, to characterize when the algorithm is guaranteed to work.

3.2.4. Cost estimation: the base-demand algorithm

Given the overhead of the alert-message mechanism, one would prefer to minimize its use. To do so requires forestalling emergencies by making sound, conservative estimates of future cash needs. Further, if most objects maintain substantial cash reserves, alert messages need not propagate far to reach a source of funds.

A renter might seek to determine its cash needs for the next cycle by multiplying the last cycle's expenses by a safety margin. Though initially plausible, this approach is unstable in the presence of cyclic client-consultant relationships: objects request money to build up safety margins, and these requests become expenses for their clients; when propagated around a loop, safety-margin multipliers cause an exponential explosion in the cash reserves and requests.

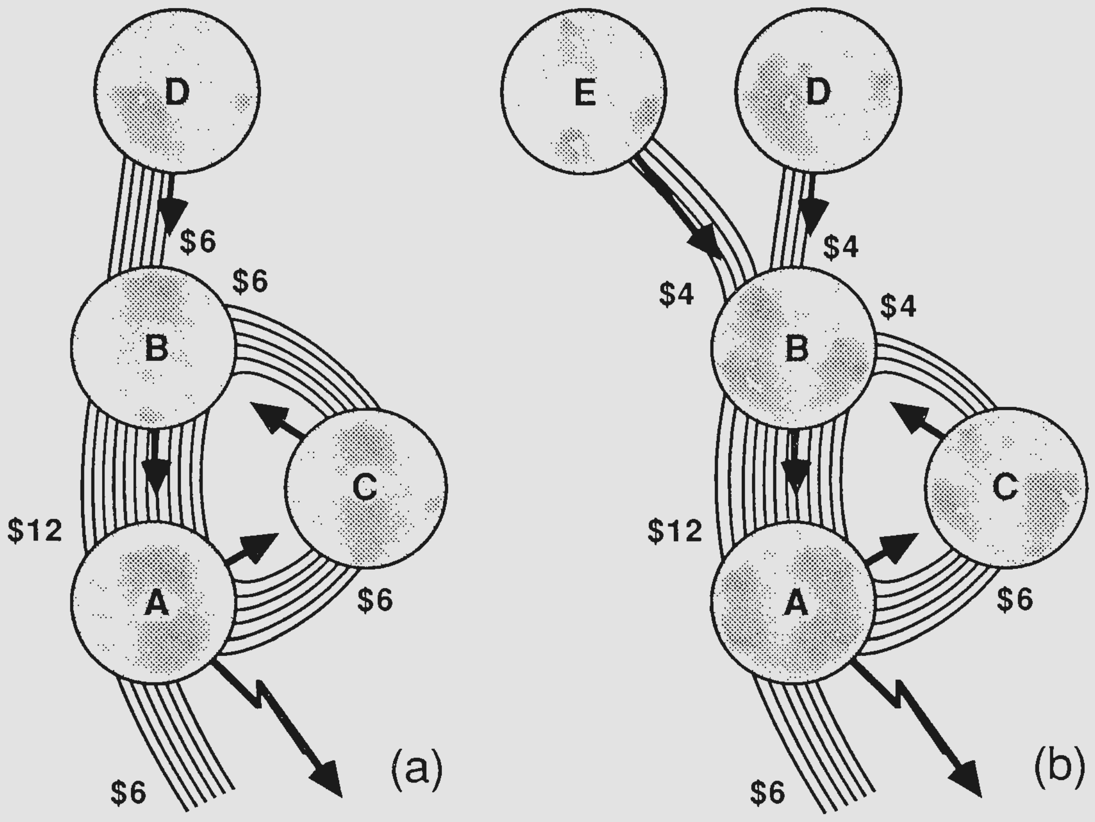
The essential idea of the base-demand algorithm is to circulate estimated-cost information to aid planning, and to do so independently of the more irregular and opportunistic circulation of money. This algorithm operates in parallel with the retainer-fee and alert algorithms just described.

With its first retainer request, R forwards to each client R's rentalPeriod (the interval until the end of R's next rent period) and a baseDemandShare equal to R's totalBaseDemand times S_{jt}. When R is newly created or retains no consultants, R's totalBaseDemand equals R's rent per unit time; otherwise it equals R's rent per unit time plus the sum of the baseDemandShares reported by R's consultants. Each renter stores a table of its consultant's baseDemandShares and rentalPeriods. A consultant's *demand chunk* is defined as the product of the consultant's reported rent period and baseDemandShare.

A candidate standard for the adequacy of a desiredBalance is that it call for enough cash on hand to eliminate shortfalls during any rent period, in a steady-state system. This requires determining an upper bound for the rent and retainer requests that may arrive and adding the resulting value to the alertReserve discussed above. One such upper bound consists of R's totalBaseDemand times R's rent period, to account for average expenses, plus the sum of all R's consultants' demand chunks, to account for a worst-case peak in demand (in which, for example, a set of long-period renters all charge retainer fees during one of R's shorter rent periods).

3.2.4.1. Analysis

The dynamic behavior of this system may be visualized in terms of a physical model in which each object's rent obligations are a source of "demand flux lines." When a new renter is introduced in a system with uniform rent periods, the demand-flux lines stemming from its



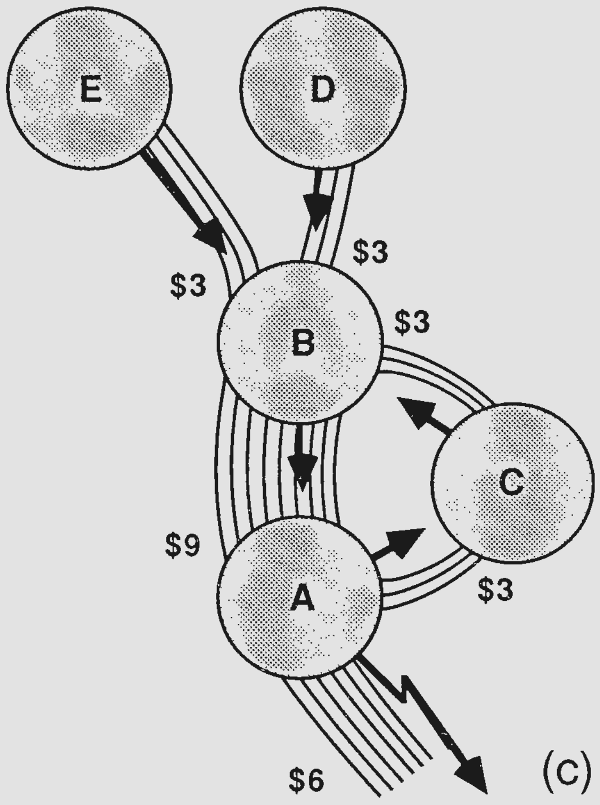


Figure 6: Base demand flux lines.

Straight arrows represent clients pointing to (retaining) consultants; the broken arrow represents A's rent payments. Each curved flux line represents a reported base demand of one dollar per rent period (flux line information propagates in the direction opposite the arrows). These diagrams assume that all clients estimate identical usage. Only flux lines originating with A's rent are shown. (Total fluxes are linear superpositions of individual fluxes.)

*6(**a**) shows the equilibrium state of A's flux lines when A is charged $6 rent. 6(**b**) shows a non-equilibrium state that results after E begins pointing at B, splitting its fee-payment with D and C (which has not yet propagated the change). 6(**c**) shows the new equilibrium, approached asymptotically in this looped case.*

rent extend one consultant-to-client step per rent period until they end in a non-retained funds source—that is, in an entity that pays its obligations out of earnings, capital, and so forth. Where a retained consultant has several clients, the bundle of lines splits but conserves total flux. When an established renter disappears, its associated flux lines suffer a wave of termination, propagating at the same speed.

Figure 6 illustrates several states in a system before and after a new client begins pointing at a looped structure. Where pointing relationships loop, but some pointers enter from the outside (as shown), a certain fraction of demand-flux escapes in each circuit of the loop. This gives the total demand flux an exponential settling behavior in which the equilibrium totalBaseDemand values accurately predict per-cycle expenses. (Non-uniform rent periods change the speed with which lines propagate, but do not change the essential dynamics.)

3.2.4.2. Open problems: the base-demand algorithm

The base-demand algorithm propagates base demand information at an awkwardly slow rate, particularly in the presence of cyclic structures. This can sometimes make emergency cash demands (and resort to the alert algorithm) unavoidable. Better heuristics for determining cash on hand would be desirable.

With the present algorithm, objects in cyclic structures will also report biased cost numbers. In a modified version of the situation shown in Figure 6, if D's estimated use of B is arbitrarily low, then B may report an arbitrarily high base demand estimate to E, although E will actually be charged no more than the sum of A, B, and C's rent. This results in B appearing less competitive than it is. This bias is unpleasant, but at least has stable consequences: if E does decide to retain B, it will be favorably surprised, and hence will have no incentive to immediately reverse its decision.

Many of the problems of this algorithm result from objects participating in cyclic structures of which they are unaware. Finding cycles by propagating full referencing information would violate the privacy of the objects involved. An open problem is to determine how little information about reference structure can be revealed while still alleviating the above problems.

There are also problems with the incentive structure of this algorithm. Information on base demand can be viewed as an indication of the expected storage surcharge for using a consultant. Objects therefore have an incentive to attempt to gain clients by deviating from the algorithm and understating costs. This is similar to the "low-balling" problem in cost-plus contracts—companies may knowingly provide a low estimate of costs while a contractor is being selected; overruns occur later, after enough time and money have been invested in the project to prevent clients from easily switching. A more market-like alternative for estimating future costs might provide rewards and penalties that would avoid this peculiar incentive.

3.3. Applications

How practical are these algorithms for storage management? They are substantially more complex than typical garbage collection algorithms, but this complexity need not be visible to a programmer—it presents a simple interface. In terms of computational resources, though, they are substantially more expensive than typical garbage collection algorithms; this restricts their applicability. One would not use them to allocate and free cons cells in a Lisp system, but one could use them to allocate and free space for large objects—even Lisp systems themselves—which might use conventional garbage collection internally.

In general, algorithms like these will make less sense when objects are small, simple, short-lived, and mutually trusting. They will make more sense when objects are large enough to make their storage costs worth considering (in the sense of "worth the overhead of computing costs and making tradeoffs"). They will make more sense when objects are complex enough to make economic decisions, and long-lived enough for the cost of making those decisions to be amortized over a significant storage time. Finally, some form of market-based storage management seems necessary if objects coded by different groups for different purposes are to make efficient use of machine resources and each other.

Some of the flaws of these algorithms become unimportant if the consequence of evicting an object is merely clearing a copy of it from a local cache or migrating it to a different machine or a different form of long-term storage. If failure to pay rent does not destroy an object, then delays in alert processing can no longer threaten program correctness. Further, large objects are more often candidates for migration than for deletion. Information of the sort circulated by the dividend and base-demand algorithms can help to tune local working-sets of objects in distributed open systems.

4. Initial strategies for trust

Many of the above algorithms make strategic sense only if one object can trust another object to follow them. For example, there are direct financial incentives to embezzle funds or misreport earnings by violating the dividend algorithm, and there are market-share incentives to produce falsely low cost estimates by violating the base-demand algorithm. Further, improper market intelligence (who is using what services?) can be gleaned by comparing alert-ID values arriving via different consultants. Thus, one needs what may be called initial strategies for trust.

The simplest strategy is for an object to trust whatever existing objects it is initially instructed to trust. This need not lead to great inflexibility or put a great burden on the programmer. Standard initial market strategies for resource management can be provided by a programming environment. In one kind of implementation, a wide range of objects will use instances of the same, small set of initial-strategy objects; these objects will recognize and trust each other, and will be able to interact with other objects in ways that do not assume trust. (Unforgeable identities are an essential foundation for trust.) Thus, use of standard initial strategies can itself be an initial strategy for trust.

Other means of building trust are discussed in [II]. They include creating or noticing situations having the characteristics of indefinitely-iterated prisoner's dilemma games [10] (see also the discussion in [I]), use of posted bonds, use of positive-reputation systems, and use of behavior-certification agencies.

5. Probabilistic cash flows

As noted in the discussion of accounting overhead in the dividend algorithm, the incentive structure of an algorithm (in the absence of risk aversion) is determined by its average *expected* payoffs, which can deviate from its actual payoffs on any given occasion. This principle has general applicability.

5.1. Processor accounting

The overhead of the escalator algorithm may be acceptable at the scale of, say, tasks in the Mach operating system [IV], but not at the finer-grained level of Mach threads, Actor tasks [11], or FCP processes [V]. Scheduling of light-weight processes like these might best be handled by a simple round-robin scheduler, which itself buys time through an auction house. How might these light-weight processes be charged so as to subject them to price incentives and compensate the round-robin process for the time it buys—all at low overhead? One approach is to use probabilistic charging: at random, uniformly-distributed times (a Poisson process with mean interarrival time T), note which light-weight process is currently running and charge its sponsoring account T times the current price of processor time. On the average, the round-robin process receives the market price for time; on the average, each light-weight process pays it. And yet on a typical occasion, a light-weight process will run without being charged, and hence without accounting overhead.

5.2. Gambling

A different kind of probabilistic cash flow is gambling, wagering money on a chance event. This too has its place.

Consider an object which has just received an alert message asking for more money than it can pay or raise though retainer-fee requests. Sending an alert message may be expensive, in terms of direct communication costs and costs imposed on clients. It is an elementary result of decision analysis [12] that when $X\%$ more money has over $X\%$ more utility, for some value of X (which requires that the utility-*vs.*-money curve somewhere be concave upwards) there exists a fair bet (or one with a small "house percentage") that is rationally worth taking. This can be the case both in alert processing and elsewhere in an agoric system.

To illustrate the principle (albeit with absurd numbers), assume that an object has a balance of $50 and receives an alert message demanding $100. Assume further that the object has 10 clients, and that transmitting an alert costs $1 per message. If the object simply alerts its clients and then pays its bill, it will pay a total of $110. If, however, the object gambles the $50 in a fair bet on a double-or-nothing basis, its expected net payment will be half the net payment that will result if the gamble is won (1/2 × $50) plus half the net payment that will result if the gamble is lost, (1/2 × ($50 + $100 + $10)). This equals $105, for an expected savings of $5. Similar bets can be profitable, so long as the house percentage amounts to less than $5. Thus, gambling might profitably be made part of a market strategy for alert processing.

One can predict that market forces will favor the emergence of rational gambling in agoric systems. To provide gambling services, one expects to see lottery objects with substantial cash reserves. These will accept payments of X units of currency with a request for a greater sum Y, and return Y with a probability slightly less than X/Y.

5.3. Insurance

Another (and more respectable) form of gambling is insurance, or risk pooling. This can be based on a form of trust that will arise naturally in an agoric system.

A set of objects sharing a single program (code, script, class) is like a set of organisms sharing a single genome. It is an elementary result of evolutionary theory [13] that the genes of such organisms (in, say, a colony) will be selected for complete altruism among "individuals". And indeed, colonial polyps often share digestive tracts, and thus all their food.

Objects sharing a script can likewise (with full trust) offer to share cash reserves, in effect insuring one another against temporary shortages and expensive alert processing. In insurance terms, the shared incentives of these objects eliminate the problem of "moral hazard", that is, of insured entities taking uneconomic risks because "the insurance company will pay for any losses". Here, objects care as much about the "insurance company" as about themselves (more accurately, "evolutionary pressures will favor those objects which behave in a manner that can be regarded as 'caring' in this way"). Objects of types which abuse this mechanism to prevent proper garbage collection will in general have higher costs and lose in price competition. This is a case in which Hofstader's "superrationality" [14] and Genesereth's "common behavior assumption" [15] will apply.

6. Conclusions

This paper has explored mechanisms for the allocation of processor time and storage that are compatible both with programming practice and with market mechanisms. Processor scheduling through an auction process yields a flexible, decentralized priority system, allowing a variety of strategies that make tradeoffs involving the speed, certainty, and cost of service. Storage can be managed through auctioning of rental space and decentralized networks of client-consultant relationships. This yields a distributed garbage collection algorithm able both to collect unreferenced loops that cross trust boundaries and to accumulate rough price information to guide economic decisions regarding, for example, local caching in distributed systems.

Some of these algorithms (*e.g.,* for processor scheduling) have per-decision costs comparable to those of non-market mechanisms in current use; others have costs that are much greater. In general, these costs will be acceptable for objects of sufficient size and processes of sufficient duration. The question of the appropriate scale at which to apply market mechanisms can be addressed by additional study but will best be addressed by experience in actual computational markets. The proposals made here can doubtless be improved upon; they are merely intended to illustrate some of the issues involved in incentive engineering for computational markets, and to provide a starting point for discussion and design. Any advances toward lower costs, greater effectiveness, and better incentive structures will shift tradeoff points in favor of finer-grained application of market mechanisms.

Even heavy overhead costs would leave intact a solid case for market mechanisms in computation. This case rests on the value of doing the right thing (or something like it) with some overhead costs, rather than doing something blatantly wrong with polished efficiency. And when finding the right thing to do requires cooperation, competition, and freewheeling experimentation, the value of decentralized systems with market accountability becomes very great indeed.

Appendix: code for the alert algorithm

The alert algorithm is the most procedurally intricate of the initial strategies described here, hence it is the one least suited to description in English. It is documented here by code written in the programming language FCP [V,16]; this code has not been run. To facilitate object-oriented programming, we are using a form of syntactic sugar known as "keyword terms" [17]. A keyword term can be distinguished from the familiar positional term by use of curly braces instead of parentheses. The arguments of a keyword term are identified by the keyword to the left of the colon instead of by position. All unmentioned keywords are considered to be associated with unbound variables. The keyword term "foo{KTerm but bar:a, baz:b}" is identical to the keyword term "KTerm" except that "bar" is associated with "a" and "baz" is associated with "b". Keyword terms can be efficiently translated into positional terms.

```
% mem
    % Alert

    % Not Idle
mem([Msg | Self], State) :-
    alert{alertID:ID, alerter:Alerter} = Msg,
    state{stateName:SName, alertID:ID} = State,
    SName =\= idle |
    Alerter = [refusal{alertID:ID}],
    mem(Self?, State).

    % IDs differ
mem([Msg | Self], State) :-
    alert{alertID:ID1} = Msg,
    state{stateName:SName, alertID:ID2} = State,
    SName =\= idle,
    ID1 =\= ID2 |
    tryToPayAlert(Msg, Self, NewSelf, State,
            NewState),
    mem(NewSelf?, NewState?).

    % Idle

    % alertQ empty
mem([Msg | Self], State) :-
    alert{} = Msg,
    state{stateName:idle, alertQ:[]} = State |
    tryToPayAlert(Msg, Self, NewSelf, State,
            NewState),
    mem(NewSelf?, NewState?).

    % alertQ non-empty
mem(Self, State) :-
    state{stateName:idle, alertQ:[AlertMsg |
            AlertMsgs], clients:Clients} = State |
    alert{alertID:ID, alerter:Alerter, amount:Amount} =
            AlertMsg,
    alertClients(Clients, alert{AlertMsg but
            alerter:Self1}, NumClients, NewClients),
    NewState = state{State but
            stateName:tryingToPayAlert, alertID:ID,
            alertQ:AlertMsgs, count:NumClients?,
            clients:NewClients?},
    merge(Self?, Self1?, NewSelf),
    mem(NewSelf?, NewState?).

    % Cancel

    % Trying to pay rent
mem([cancel{alertID:ID} | Self], State) :-
```

```
    state{stateName:tryingToPayRent, alertID:ID} =
            State |
    mem(Self?, State).

    % Trying to pay alert
mem([cancel{alertID:ID} | Self], State) :-
    state{stateName:tryingToPayAlert, alertID:ID,
            clients:Clients} = State |
    cancelClients(Clients, cancel{alertID:ID},
            NewClients),
    NewState = state{State but stateName:idle,
            clients:NewClients},
    mem(Self?, NewState?).

    % Waiting to be thanked
mem([cancel{alertID:ID} | Self], State) :-
    state{stateName:waitingToBeThanked, alertID:ID}
            = State |
    mem(Self?, State).

    % Ids differ
mem([cancel{alertID:ID1} | Self], State) :-
    state{stateName:SName, alertID:ID2, alertQ:Q} =
            State,
    SName =\= idle,
    ID1 =\= ID2 |
    forgetAlert(Q?, ID1, NewQ),
    NewState = state{State but alertQ:NewQ?},
    mem(Self?, NewState?).

    % Idle
mem([cancel{} | Self], State) :-
    state{stateName:idle, alertQ:[]} = State |
    mem(Self?, State).

    % Thank you

    % Trying to pay rent
mem([Msg | Self], State) :-
    thankYou{alertID:ID} = Msg,
    state{stateName:tryingToPayRent, alertID:ID} =
            State |
    error(Msg),
    mem(Self?, State).

    % Trying to pay alert
mem([Msg | Self], State) :-
    thankYou{alertID:ID} = Msg,
    state{stateName:tryingToPayAlert, alertID:ID} =
            State |
    error(Msg),
    mem(Self?, State).

    % Waiting to be thanked
mem([Msg | Self], State) :-
    thankYou{alertID:ID} = Msg,
    state{stateName:waitingToBeThanked, alertID:ID,
            payer:Payer} = State |
    Payer = [Msg],
    NewState = state{State but stateName:idle},
    mem(Self?, NewState?).

    % Ids differ
mem([thankYou{alertID:ID1} | Self], State) :-
    state{stateName:SName, alertID:ID2} = State,
    SName =\= idle,
    ID1 =\= ID2 |
    mem(Self?, State).

    % Idle
mem([thankYou{} | Self], State) :-
    state{stateName:idle} = State |
    mem(Self?, State).

    % Reimbursement

    % Trying to pay rent
mem([Msg | Self], State) :-
    reimbursement{alertID:ID} = Msg,
    state{stateName:tryingToPayRent, alertID:ID} =
            State |
    error(Msg),
    mem(Self?, State).

    % Trying to pay alert
mem([Msg | Self], State) :-
    reimbursement{alertID:ID} = Msg,
    state{stateName:tryingToPayAlert, alertID:ID} =
            State |
    error(Msg),
    mem(Self?, State).

    % Waiting to be thanked
mem([Msg | Self], State) :-
    reimbursement{alertID:ID} = Msg,
    state{stateName:waitingToBeThanked, alertID:ID,
            payer:Payer} = State |
    Payer = [Msg],
    NewState = state{State but stateName:idle},
    mem(Self?, NewState?).

    % Ids differ
mem([Msg | Self], State) :-
    reimbursement{alertID:ID1, check:Check} = Msg,
    state{alertID:ID2} = State,
    ID1 =\= ID2 |
    deposit(Check?, State?, NewState),
    mem(Self?, NewState?).

    % Idle
mem([Msg | Self], State) :-
    reimbursement{check:Check} = Msg,
    state{stateName:idle} = State |
    deposit(Check?, State?, NewState),
    mem(Self?, NewState?).

    % Payment

    % Trying to pay rent
mem([Msg | Self], State) :-
    payment{alertID:ID, check:Check, payer:Payer} =
            Msg,
    state{stateName:tryingToPayRent, alertID:ID,
            clients:Clients} = State |
    payRent(Check?, State?, State1),
    Payer = [thankYou{alertID:ID} | Payer1],
    creditPayer(Payer1, Check?, State1?, State2),
    cancelClients(Clients, cancel{alertID:ID},
            NewClients),
    NewState = state{State2 but stateName:idle,
            clients:NewClients},
    mem(Self?, NewState?).

    % Trying to pay alert
mem([Msg | Self], State) :-
    payment{alertID:ID, check:Check, payer:Payer} =
            Msg,
    state{stateName:tryingToPayAlert, alertID:ID,
            alerter:Alerter, clients:Clients} = State |
    creditPayer(Payer, Check?, State?, State1),
    Alerter = [payment{Msg but payer:Self1}],
    cancelClients(Clients, cancel{alertID:ID},
            NewClients),
    NewState = state{State1? but
            stateName:waitingToBeThanked,
            payer:Payer, clients:NewClients},
    mem(Self?, NewState?).

    % Waiting to be thanked
mem([Msg | Self], State) :-
    payment{alertID:ID, check:Check, payer:Payer} =
            Msg,
    state{stateName:waitingToBeThanked, alertID:ID}
            = State |
    Payer = [reimbursement{alertID:ID,
            check:Check}],
    mem(Self?, State).

    % ids differ
mem([Msg | Self], State) :-
    payment{alertID:ID1, check:Check, payer:Payer}
            = Msg,
    state{stateName:SName, alertID:ID2} = State,
    SName =\= idle,
    ID1 =\= ID2 |
    Payer = [reimbursement{alertID:ID,
            check:Check}],
    mem(Self?, State).

    % Idle
mem([Msg | Self], State) :-
    payment{alertID:ID, check:Check, payer:Payer} =
            Msg,
    state{stateName:idle} = State |
    Payer = [reimbursement{alertID:ID,
            check:Check}],
    mem(Self?, State).

    % Refusal

    % Trying to pay rent

    % count = 0
mem(Self, State) :-
    state{stateName:tryingToPayRent, count:0} =
            State |
    liquidate(Self, State).

    % count > 0
mem([refusal{alertID:ID} | Self], State) :-
    state{stateName:tryingToPayRent, alertID:ID,
            count:Count} = State,
    Count > 0 |
    NewCount := Count - 1,
    NewState = state{State but count:NewCount?},
    mem(Self?, NewState?).

    % Trying to pay alert

    % count = 0
mem(Self, State) :-
    state{stateName:tryingToPayAlert, alertID:ID,
            count:0, alerter:Alerter} = State |
    Alerter = [refusal{alertID:ID}],
    NewState = state{State but stateName:idle},
    liquidate(Self, NewState?).

    % count > 0
mem([refusal{alertID:ID} | Self], State) :-
    state{stateName:tryingToPayAlert, alertID:ID,
            count:Count} = State,
    Count > 0 |
    NewCount := Count - 1,
    NewState = state{State but count:NewCount?},
    mem(Self?, NewState?).

    % Waiting to be thanked
mem([refusal{alertID:ID} | Self], State) :-
    state{stateName:waitingToBeThanked, alertID:ID}
            = State |
    mem(Self?, State).

    % Ids differ
mem([refusal{alertID:ID1} | Self], State) :-
    state{stateName:SName, alertID:ID2} = State,
    SName =\= idle,
    ID1 =\= ID2 |
    mem(Self?, State).

    % idle
mem([refusal{} | Self], State) :-
    state{stateName:idle} = State |
    mem(Self?, State).
```

```
% Other predicates

% tryToPayAlert
tryToPayAlert(AlertMsg, Self, NewSelf, State,
        NewState) :-
    alert{amount:Amount} = AlertMsg,
    collectRetainer(Amount?, State?, State1, Check,
        Ok),
    tryToPayAlert1(Ok?, Check?, AlertMsg, Self,
        NewSelf, AlertsForQ),
    state{alertQ:Q} = State1?,
    append(AlertsForQ?, Q?, NewQ),
    NewState = state{State1? but alertQ:NewQ?}.

tryToPayAlert1(true, Check, AlertMsg, Self, NewSelf,
        []) :-
    alert{alertID:ID, alerter:Alerter} = AlertMsg,
    Alerter = [payment{alertID:ID, check:Check,
        payer:Self1}],
    merge(Self?, Self1?, NewSelf).

tryToPayAlert1(false, Check, AlertMsg, Self,
        NewSelf, [Alert]).

% alertClients
alertClients([], AlertMsg, 0, []) :-
    alert{alerter:[]} = AlertMsg.

alertClients([Client | Clients], AlertMsg, NumClients,
        [NewClient | NewClients]) :-
    alert{alerter:Alerter} = AlertMsg,
    Client = [alert{AlertMsg but alerter:Alerter1} |
        NewClient?],
    alertClients(Clients, alert{AlertMsg but
        alerter:Alerter2}, NCMinus1, NewClients),
    NumClients := 1 + NCMinus1,
    merge(Alerter1?, Alerter2?, Alerter).

% cancelClients
cancelClients([], CancelMsg, []).

cancelClients([Client | Clients], CancelMsg,
        [NewClient | NewClients]) :-
    Client = [CancelMsg | NewClient?],
    cancelClients(Clients, CancelMsg, NewClients).

% forgetAlert
forgetAlert([], ID, []).

forgetAlert([AlertMsg | Q], ID, Q) :-
    alert{alertID:ID, alerter:[]} = AlertMsg | true.

forgetAlert([AlertMsg | Q], ID1, [AlertMsg | NewQ?]) :-
    alert{alertID:ID2} = AlertMsg,
    ID1 =\= ID2 |
    forgetAlert(Q, ID1, NewQ).
```

Acknowledgments

Note: see the paper "Markets and Computation: Agoric Open Systems" in this book [II] for general discussion, acknowledgments, and comparison with other work.

References

Papers referenced with roman numerals can be found in the present volume:

Huberman, Bernardo (ed.), *The Ecology of Computation* (Elsevier Science Publishers/North-Holland, 1988).

[I] Miller, Mark S., and Drexler, K. Eric, "Comparative Ecology: A Computational Perspective", this volume.

[II] Miller, Mark S., and Drexler, K. Eric, "Markets and Computation: Agoric Open Systems", this volume.

[III] Liskov, Barbara, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", this volume.

[IV] Rashid, Richard F., "From RIG to Accent to Mach: The Evolution of a Network Operating System", this volume.

[V] Kahn, Kenneth, and Miller, Mark S., "Language Design and Open Systems", this volume.

[1] Lieberman-Hewitt Algorithm: Lieberman, Henry, and Hewitt, Carl, "A Real-Time Garbage Collector Based on the Lifetimes of Objects", in: *Communications of the ACM* (June 1983) 26, 6, pp.419-429.

[2] Quarterman, John S., Silberschatz, Abraham, Peterson, James L., "4.2BSD and 4.3BSD as Examples of the UNIX System", in: *ACM Computing Surveys* (December 1985) Vol. 17 No. 4 pp.379–418.

[3] Rees, Jonathan A., and Adams, Norman I., IV, "T: a Dialect of Lisp or, Lambda: The Ultimate Software Tool", in: *Proceedings of*

the 1982 ACM Symposium on Lisp and Functional Programming (August 1982).

[4] Bishop, Peter B., *Computers with a Large Address Space and Garbage Collection* (MIT, Cambridge, MA, May 1977) MIT/LCS/TR-178.

[5] Smith, Vernon L., "Experimental Methods in the Political Economy of Exchange", in: *Science* (10 October 1986) Vol. 234, pp.167–173.

[6] Friedman, Daniel, "On the Efficiency of Experimental Double Auction Markets", in: *American Economic Review* (March 1984) Vol. 24, No. 1, pp. 60-72.

[7] Theriault, D., *Issues in the Design and Implementation of Act 2* (MIT AI Lab, Cambridge, MA., 1983) AI-TR-728.

[8] Kornfeld, William A., "Using Parallel Processing for Problem Solving" (MIT AI Lab, Cambridge, MA, 1979) MIT-AI-561.

[9] Ungar, David Michael, *The Design and Evaluation of a High Performance Smalltalk System* (MIT Press, Cambridge, MA, 1987).

[10] Axelrod, Robert, *The Evolution of Cooperation* (Basic Books, New York, 1984).

[11] Agha, Gul, *Actors: A Model of Concurrent Computation in Distributed Systems* (MIT Press, Cambridge, MA, 1986).

[12] Raffia, Howard, *Decision Analysis: Introductory Lectures on Choices under Uncertainty* (Addison-Wesley, Reading, MA, 1970).

[13] Dawkins, Richard, *The Selfish Gene* (Oxford University Press, New York, 1976).

[14] Hofstadter, Douglas R., "Dilemmas for Superrational Thinkers, Leading Up to a Luring Lottery", in: *Metamagical Themas: Questing for the Essence of Mind and Pattern* (Basic Books, New York, 1985) pp. 739-755.

[15] Genesereth, M. R., Ginsberg, M. L., and Rosenschein, J. S., *Cooperation without Communication* (1984) HPP Report 84-41.

[16] Shapiro, Ehud, (ed.), *Concurrent Prolog: Collected Papers* (MIT Press, Cambridge, MA, 1987) in press.

[17] Hirsh, Susan, Kahn, Kenneth M., and Miller, Mark S., *Interming: Unifying Keyword and Positional Notations* (Xerox PARC, Palo Alto, CA, 1987) in press.