An Object-Oriented Framework for Modular Resource Management

Carl A. Waldspurger William E. Weihl
Digital Systems Research Center
Palo Alto, CA 94301 USA
{caw,weihl}@pa.dec.com

Abstract

We present a flexible object-oriented framework for specifying modular resource management policies in concurrent systems. The framework generalizes the basic abstractions we originally developed for lottery scheduling [16]. It is independent of the underlying proportional-share scheduler; a variety of probabilistic and deterministic algorithms can be used, including a min-funding revocation algorithm that we introduce for space-shared resources. The framework supports diverse resources and policies, including both proportional shares and guaranteed reservations. A repayment mechanism prevents allocation distortions caused by transfers of resource rights. Key framework concepts are analogous to features of object-oriented languages.

1. Introduction

Effective resource management requires knowledge of both user preferences and application-specific performance characteristics. Unfortunately, traditional operating systems centrally manage machine resources within the kernel [5], affording clients only crude control through *ad hoc* interfaces that inhibit modularity. For example, dynamic priority schedulers are difficult to understand, provide poor control over service rates, and violate modular abstraction principles [2, 6, 7, 16].

This paper advocates a radically different approach to computational resource management. *Tickets* are first-class objects that represent resource rights, allowing clients to express a wide range of resource management policies. *Currencies* abstract collections of tickets to permit modular composition, allowing policies to be expressed conveniently at various levels of abstraction. Resource rights vary smoothly with easily-understood ticket allocations, simplifying the specification of custom policies.

The framework described here generalizes the basic abstractions we originally developed for lottery scheduling [16]: it is independent of the underlying proportional-share scheduling algorithm, and it incorporates a novel *repayment* mechanism to prevent distortions of service rates caused by ticket transfers. We also describe a novel deterministic scheduling algorithm – *min-funding revocation* – for space-shared resources, sketch several extensions to the basic framework, and present numerous examples that demonstrate the framework's versatility.

2. Framework

Our framework consists of two objects: *tickets*, which encapsulate resource rights, and *currencies*, which flexibly name, share, and protect collections of tickets.

2.1. Tickets

Resource rights are encapsulated by first-class objects called *tickets*. Tickets can be issued in different amounts; a single physical ticket may represent any number of logical tickets. Tickets are owned by *clients* that consume resources. A client is *active* while it is competing to acquire more resources. An active client is entitled to consume resources at a rate proportional to its ticket allocation. Thus, ticket allocations determine service rates for timeshared resources, and storage capacity for space-shared resources.

In general, tickets represent *relative* rights that depend on the total number of tickets contending for a resource. Client allocations degrade gracefully in overload situations, and active clients proportionally benefit from extra resources when some allocations are underutilized. In the worst case, an active client's share is proportional to its share of tickets in the system. Thus, *absolute* rights can be specified by fixing the total supply of tickets, ensuring that each ticket represents a guaranteed minimum resource fraction.

2.2. Ticket Transfers

The most basic ticket operation is a direct redistribution of resource rights via a *ticket transfer* between clients. Transfers are particularly useful in any situation where one

client blocks waiting for other clients: the waiting client can transfer some or all of its tickets to the others, allowing them to acquire more resources and hence complete faster.

For example, the client of an RPC can transfer tickets to the server, which then executes with the resource rights of the client, and returns those rights during the RPC reply. Similarly, a client waiting to acquire a lock can temporarily transfer tickets to the current lock owner, solving the conventional priority inversion problem in a manner similar to priority inheritance [12]. Unlike priority inheritance, transfers from multiple clients are additive. A client can also split transfers across several clients on which it is waiting (*e.g.*, a writer waiting on multiple readers).

Ticket transfers are capable of specifying *any* ticket-based resource management policy, since an arbitrary distribution of tickets to clients can be effected. However, transfers are often too low-level to be convenient, since they impose a *conservation* constraint: tickets can be redistributed, but they cannot be created or destroyed.

2.3. Transfer Repayment

A temporary ticket transfer can be viewed as either a *donation* or a *loan* of the resources acquired using the ticket during the transfer period. For example, when an RPC client transfers tickets to a server, it is appropriate to view the transfer as a donation, since the resources acquired with the client's tickets are used on its behalf. However, tickets transferred from a client blocked on a lock to its owner are better viewed as a loan that must be *repaid*, since the lock owner is not computing on behalf of the client.

We simulated a system using *stride scheduling* to implement proportional sharing for both processor time and lock access, and studied ticket transfers with and without repayment. Without repayment, computation rates are distorted to favor lock holders; with repayment, they closely approximate the specified rates [15].

In economic terms, a ticket behaves like a constant monetary income stream. We are currently investigating the general problem of allowing limited *accumulation* of resource rights without sacrificing accuracy guarantees.

2.4. Ticket Inflation and Deflation

Ticket inflation and *deflation* are alternatives to explicit ticket transfers. Client resource rights can be escalated by creating more tickets, inflating the overall supply of tickets. Similarly, client resource rights can be reduced by destroying tickets, deflating the supply. Inflation and deflation are useful among mutually trusting clients and are often easier to use than transfers, since they permit resource rights to be reallocated without explicitly reshuffling tickets among clients. For example, a process can allocate resource rights equally to subprocesses simply by creating and assigning a fixed number of tickets to each subprocess, and destroying the tickets owned by each subprocess when it terminates.

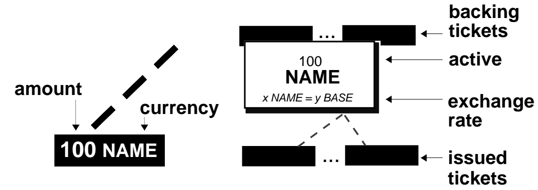

Figure 1. Ticket and Currency Objects.

Uncontrolled ticket inflation is dangerous, since a client can monopolize a resource by creating a large number of tickets. Viewed from an economic perspective, inflation is a form of theft, since it devalues the tickets owned by all clients. Because inflation can violate desirable modularity properties, it must be either prohibited or strictly controlled.

2.5. Currencies

The desirability of inflation and deflation hinges on *trust*. Trust implies permission to appropriate resources without explicit authorization. A *currency* defines a modular *trust boundary* that contains the effects of ticket inflation. Each ticket is extended to include a currency in which it is denominated, allowing resource rights to be expressed in units that are local to each group of mutually trusting clients. Each currency should maintain permissions (*e.g.*, access control lists) that determine which clients have the right to create and destroy tickets denominated in that currency.

Figure 1 depicts key aspects of ticket and currency objects. A ticket consists of an amount denominated in some currency, denoted by *amount.currency*. A currency has a unique name, is *funded* by a set of *backing tickets* denominated in more primitive currencies, and maintains a list of issued tickets and the total number that are active. Inflation is locally contained by maintaining an *exchange rate* between each currency and a common *base* currency that is conserved. Tickets denominated in different currencies are compared by first converting them into the base currency.

One useful currency configuration is a hierarchy of currencies. In general, currency relationships may form any DAG. Figure 2 depicts an example currency graph; each user and task has an associated currency. Two users, Alice and Bob, are competing for computing resources. Bob's *task2* is blocked on a lock held by Alice's *task1*, and transfers its funds to expedite the lock's release. The current values in base units for the runnable tasks are *task1* = 4000, and *task3* = 1000.

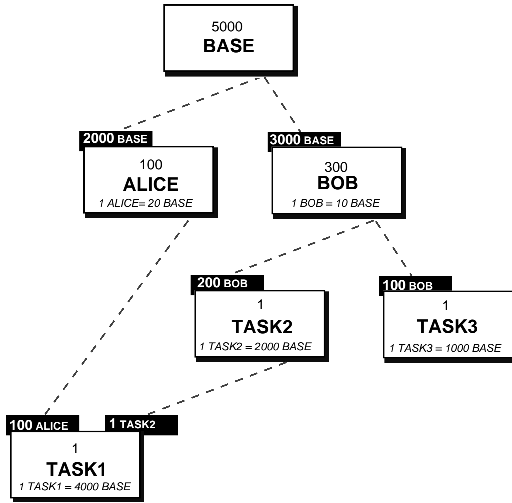

Figure 2. Example Currency Graph.

2.6. Currencies Resemble Classes

Currency abstractions for resource rights resemble data abstractions for data objects. A data abstraction defines an *abstraction barrier* between an abstract data type and its underlying representation [9] that both hides and protects the representation. Normally, access is allowed only through exported operations; however, the operations themselves can manipulate the representation. A currency defines a resource management abstraction barrier that provides similar properties for resource rights: clients are generally untrusted, and only those with explicit permission are allowed to distribute resource rights within a currency.

Resource-right relationships structured by currencies also resemble object relationships structured by classes in object-oriented languages with multiple inheritance. A class inherits its behavior from a set of superclasses; similarly, a currency inherits its resource rights from a set of backing tickets in more primitive currencies. Tickets, which are instances of currencies, are also similar to objects, which are instances of classes. However, issuing a new ticket dilutes the value of all existing tickets denominated in a currency, while the instances of a class need not affect one another.

3. Proportional-Share Algorithms

The general framework presented above can be implemented by any proportional-share scheduling algorithm. However, efficient low-level support for dynamic changes in the set of active clients and their allocations is required to implement higher-level framework operations. Appropriate algorithms also depend on the resource to be allocated.

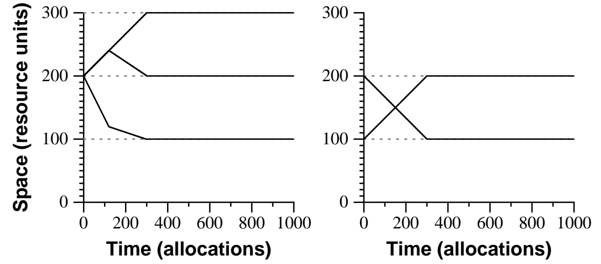

Figure 3. Min-Funding Revocation.

3.1. Time-Shared Resources

For time-shared resources, such as processor time, I/O bandwidth, and lock access, it is impossible to achieve ideal behavior due to quantization. A key characteristic that distinguishes different algorithms is accuracy. Our randomized *lottery scheduling* algorithm has expected $O(\sqrt{n_a})$ error after n_a allocations [16]. Deterministic algorithms based on *virtual time* have greatly improved accuracy. A virtual clock is effectively associated with each client, and ticks at a rate inversely proportional to the client's ticket allocation. A resource allocation is performed by selecting the client with the minimum virtual clock time.

Our *stride scheduling* approach generalizes virtual-time flow-control algorithms designed for networks [3, 17, 11] for use with other time-shared resources; hierarchical stride scheduling exhibits $O(\lg n_c)$ error for n_c clients, independent of n_a [15]. Optimal deterministic algorithms, such as *EEVDF* [13] and *WF*^{2}*Q* [1], bound the error to a *single* quantum by performing an additional check that makes clients which are ahead of schedule ineligible for the current allocation. The precision of these algorithms makes them particularly attractive for supporting real-time clients.

3.2. Space-Shared Resources

We have also developed mechanisms for allocating divisible space-shared resources, such as blocks in a filesystem buffer cache or resident VM pages [15]. Dynamic space-sharing is based on resource revocation. When a client demands more space, a replacement algorithm selects a victim client that relinquishes some of its previously allocated space. Our deterministic *min-funding revocation* algorithm [15] dynamically converges to proportional-share allocations approximately twice as fast as our earlier *inverse lottery* approach [16].

Performing a revocation is simple: a resource unit is revoked from the client expending the fewest tickets per resource unit, compared with other clients. As its name suggests, *min-funding revocation* also has a clear economic interpretation. The number of tickets per resource unit can

be viewed as its price; revocation reallocates resource units away from clients paying a lower price to clients willing to pay a higher price. Thus, resource consumption adaptively expands and contracts as a function of current contention levels. A client with a constant number of tickets can increase its consumption when contention falls, and its consumption will decrease due to revocations when contention rises. When a client loses a resource unit due to a revocation, its ticket to resource unit ratio increases, making it more resistant to future revocations. Similar reasoning applies in the opposite direction for expansion.

Figure 3 presents simulation data showing how resource allocations evolve over time given initial ticket and space allocations. The graph on the left plots resource allocations over time for three clients with a $3:2:1$ ticket ratio, starting with equal space allocations. All clients issue allocation requests at the same rate, and their resource levels converge to the desired levels after approximately 300 allocations. The graph on the right shows space allocations over time for two clients with a $2:1$ ticket ratio. The clients start with a $1:2$ resource allocation, the reverse of their specified proportional shares. Although the second client issues allocation requests at twice the rate of the first client, resource shares still rapidly converge to the desired levels.

3.3. Framework Implementation

We have implemented the complete framework for processor timesharing in the Mach kernel, using an underlying lottery scheduler [16, 15]. The implementation employs a *lazy* strategy that defers converting ticket values into common base currency units until their values are actually needed. An alternative *eager* strategy would perform currency conversions immediately.

4. Framework Extensions

This section introduces extensions to the core framework that support connections with the external economy and differing notions of fairness.

4.1. Exhaustible Tickets

In the basic framework, tickets are not consumed when used to acquire resources. *Exhaustible tickets* generalize tickets by adding an expiration time limit; this is useful if tickets are exchanged for real money in the external economy. For example, a client could spend more money to acquire more tickets, thus obtaining the right to faster service; each ticket could be valid for a specified amount of time or usage, thus requiring each client to make periodic payments for the right to continue to use a resource.

Exhaustible tickets can be defined using several different kinds of time limits, leading to different resource pricing schemes. Assume that the price of a ticket is proportional to its value in base tickets. An exhaustible ticket based on *active competition* time can compete for a limited number of allocations, and has an economic interpretation in which the monetary price of a resource is proportional to the number of tickets competing for it. As a result, the amount paid per unit of resource consumption varies with contention, and each client pays an amount proportional to its ticket allocation during each unit of time that it is active.

A slightly different policy arises when tickets expire after a fixed period of *elapsed* time. This implements a pricing policy that charges for the *opportunity* to compete for resources, regardless of actual usage. This policy is similar to the one above based on active competition time, but prevents clients from buying tickets and then holding them in reserve. As a result, the actual monetary price for a ticket can be easily varied based on real time (*e.g.*, time-of-day), providing an additional variable to control the monetary price for a resource. This makes it easy for a service provider to limit the total number of tickets that can compete for a resource in a given time period.

Another option is to base expirations on *resource consumption* time, so that a ticket represents a fixed quantity of resource usage, consumable at a rate that depends on contention. This provides clients with more predictable monetary prices for resources, since the amount of money paid for a unit of resource no longer varies with contention (though the service rate for a given client does).

4.2. Fairness Time Scale

The basic framework assumes an *instantaneous* form of fairness in which the resource consumption rates of *active* clients are proportional to their ticket allocations. Sometimes it may be desirable to provide a *time-averaged* form of sharing based on actual resource usage, measured over some time interval. For example, if a client is temporarily inactive, a scheduler that provides time-averaged fairness would allow it to "catch up" when it becomes active. Note that a client could monopolize resources while catching up; this is not permitted by instantaneous fairness.

The framework does not require any modifications to support time-averaged fairness. However, since appropriate time-averaging intervals vary across applications, we are currently exploring efficient low-level implementation techniques to support per-currency fairness time scales.

5. Example Policies

A wide variety of policies can be specified using the general resource management framework. This section exam-

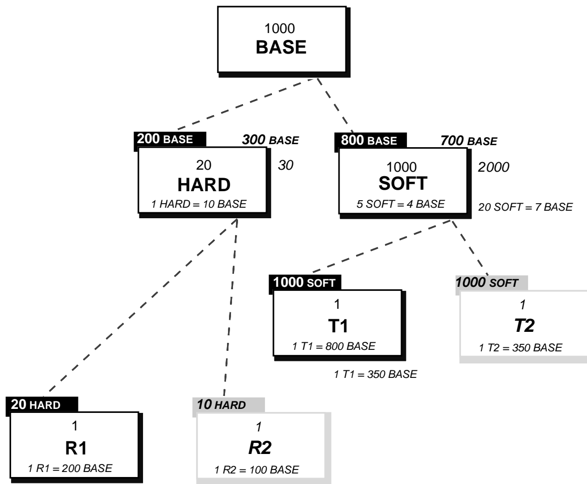

Figure 4. Timesharing and Reservations.

ines several different resource management scenarios, and demonstrates how appropriate policies can be specified.

5.1. Resource Shares

As mentioned earlier, tickets generally specify *relative* resource shares. However, if the total number of tickets in a system is fixed, then tickets also specify *absolute* resource shares. Absolute shares can be specified by issuing tickets in any *hard* currency, defined as a currency that maintains a fixed exchange rate with a conserved base currency.

5.2. Timesharing/Real-Time Integration

Currency configurations may contain both timesharing clients that adapt to changes in resource availability, and real-time clients that demand resource reservations (*e.g.*, guaranteed cycles or pinned pages). Each timesharing client is allocated a relative resource share by inflating a *soft* currency to issue tickets to the client, thus diluting the resource shares of other timesharing clients. Each real-time client is allocated an absolute resource share by issuing it new tickets in a *hard* currency, but only after adding backing tickets to the currency so that hard ticket values remain constant.

Figure 4 depicts an example configuration with both real-time and timesharing clients. Initially, *hard* funds real-time client *R1* with a guaranteed 20% reservation, and *soft* funds timesharing client *T1*. When additional clients are added (shown in gray with italic side annotations), *hard* is inflated to fund *R2* with a 10% reservation, and 100.*base* is transferred from *soft* to maintain a constant $hard : base = 1 : 10$ exchange rate. In contrast, when *soft* is inflated to fund *T2*, *T1*'s share decreases.

5.3. Progress-Based Funding

Ticket inflation and deflation provide a convenient way for concurrent clients to implement resource management policies. For example, cooperative (AND-parallel) clients can independently adjust their ticket allocations based upon application-specific estimates of remaining work (as in the Monte-Carlo example in [16, 15]). Similarly, competitive (OR-parallel) clients, such as heuristic searches or speculative computations, can independently adjust their allocations based on application-specific progress metrics.

5.4. Priority Emulation

Absolute priorities can be *approximated* by a series of currencies $c_1, \ldots, c_n$ configured such that currency c_{i-1} is backed by a single ticket in currency c_i, and a client with emulated priority i is allocated k tickets in currency c_i. Each client with priority i will be serviced k times more frequently than the set of all clients with lower priority, approximating a strict priority ordering.

5.5. Interactive Applications

Interactive systems must rapidly focus limited resources on those tasks that are currently important [4, 14]. Importance can be represented by ticket allocations, which could be controlled with a simple GUI, and also by associating tickets with the input focus to accelerate applications in response to mouse movements.

Many interactive systems, including databases and the Web, use servers to process requests from a wide variety of clients with different service demands, reflecting inherent importance or monetary premiums for better service. Tickets could specify importance, and ticket transfers would ensure that clients get the service they request.

5.6. System Administration

Engineering and scientific centers need to allocate shared resources among users and applications of varying importance [7]. Many corporations must also manage scarce computing resources, such as overloaded network firewalls.

System administrators can create currencies for different groups, which can subdivide their allocations among users autonomously. Since currency relationships need not follow a strict hierarchy, users may belong to multiple groups, and one group can even subsidize another.

5.7. Dynamic Resource Tradeoffs

Multiple heterogeneous resources must be managed concurrently. One approach is to use resource-specific tickets,

each valid only for a single resource. Alternatively, uniform tickets could be allowed to compete for any resource, permitting clients to make quantitative cost-benefit tradeoffs among different resources.

6. Related Work

As mentioned earlier, traditional operating systems provide only crude control over resource management. For example, dynamic priority schedulers, which are the dominant paradigm for managing processor time in modern operating systems, are complex, *ad hoc*, and hard to understand [2, 7, 6]. Resource rights do not vary smoothly with priorities. Priority mechanisms also violate modular abstraction principles: when separately developed modules are combined, the internal priority values in each must be exposed to understand resource allocation in the resulting system. For other resources, such as filesystem buffers, disk bandwidth, and lock access, the control – if any – is equally poor.

Real-time schedulers also lack modularity, and impose onerous restrictions on applications. However, a higher-level *processor capacity reserve* abstraction [10] provides limited control of processor usage across protection boundaries, similar to a restricted form of ticket transfers.

Fair-share schedulers allocate resources to groups or users in proportion to the number of *shares* they have been assigned, providing time-averaged fairness over long periods of time [8, 7]. Shares are typically assigned directly to individual users or groups; hierarchical share allocation has also been implemented [8]. However, shares are not treated as first-class objects, preventing the specification of more general resource management policies. Prior work on proportional-share schedulers that implement instantaneous fairness is outlined in Section 3.

7. Conclusions

In this paper we have described a new resource management framework that generalizes the abstractions we originally developed for lottery scheduling [16]. The framework, based on tickets and currencies, is simple yet flexible, is independent of the underlying proportional share scheduling algorithm, supports modular composition of custom policies, and provides an easy-to-understand relationship between user specifications and resulting allocations.

In addition to generalizing the framework, we have presented a novel *min-funding revocation* algorithm for space-shared resources that provides fast convergence to specified shares, along with a *repayment* mechanism for ticket transfers that prevents service rate distortions. We also sketched two other potential extensions – exhaustible tickets and time-averaged fairness – and presented numerous examples that demonstrate the versatility of the framework.

References

[1] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case Fair Weighted Fair Queueing. In *Proceedings of IEEE INFOCOM*, Mar. 1996.

[2] H. M. Deitel. *Operating Systems*. Addison-Wesley, Reading, MA, 1990.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, 1(1):3–26, Sept. 1990.

[4] D. Duis and J. Johnson. Improving user-interface responsiveness despite performance limitations. In *Proceedings of the Thirty-Fifth IEEE Computer Society International Conference*, pages 380–386, Mar. 1990.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Dec. 1995.

[6] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 94–105, Dec. 1993.

[7] J. L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, Aug. 1993.

[8] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, Jan. 1988.

[9] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, 1986.

[10] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: An abstraction for managing processor usage. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 129–134, Oct. 1993.

[11] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[13] I. Stoica and H. Abdel-Wahab. Earliest Eligible Virtual Deadline First: A flexible and accurate mechanism for proportional share resource allocation. Technical Report 95-22, Old Dominion University, Norfolk, VA, Nov. 1995.

[14] S. H. Tang and M. A. Linton. Pacers: Time-elastic objects. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 35–43, Nov. 1993.

[15] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. Ph.D. thesis, MIT/LCS/TR-667, MIT, Cambridge, MA, Sept. 1995.

[16] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, Nov. 1994.

[17] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.