Competitive Freshness Algorithms for Wait-free Data Objects

Peter Damaschke, Phuong Hoai Ha, and Philippas Tsigas

Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. {ptr, phuong, tsigas}@cs.chalmers.se

Abstract. Wait-free concurrent data objects are widely used in multiprocessor systems and real-time systems. Their popularity results from the fact that they avoid locking and that concurrent operations on such data objects are guaranteed to finish in a bounded number of steps regardless of the other operations interference. The data objects allow high access parallelism and guarantee correctness of the concurrent access with respect to its semantics. In such a highly-concurrent environment, where many wait-free write-operations updating the object state can overlap a single read-operation, the age/freshness of the state returned by this read-operation is a significant measure of the object quality, especially for real-time systems.

In this paper, we first propose a freshness measure for wait-free concurrent data objects. Subsequently, we model the freshness problem as an online problem and present two algorithms for it. The first one is a deterministic algorithm with asymptotically optimal competitive ratio $\sqrt{\alpha}$, where α is a function of the execution-time upper-bound of wait-free operations. The second one is a competitive randomized algorithm with competitive ratio $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$.

1 Introduction

Concurrent data objects play a significant role in multiprocessor systems, but also create challenges on consistency. In concurrent environments like multiprocessor systems, consistency of a shared data object is guaranteed mostly by mutual exclusion, a form of locking. However, mutual exclusion degrades the system's overall performance due to lock convoying, i.e. other concurrent operations cannot make any progress while the access to the shared object is blocked. Mutual exclusion also contains risks of deadlock and priority inversion. To address these problems, researchers have proposed *non-blocking algorithms* for shared data objects. Non-blocking methods do not involve mutual exclusion, and therefore do not suffer the problems that blocking can cause. Non-blocking algorithms are either lock-free or wait-free. *Lock-free* [11] algorithms guarantee that regardless of both the contention caused by concurrent operations and the interleaving of their sub-operations, always at least one operation will progress. However, there is a risk for starvation as progress of other operations could cause one specific operation to never finish. *Wait-free* [10] algorithms are lock-free and moreover they avoid starvation. In a wait-free algorithm every operation is guaranteed to finish in a limited number of steps, regardless of actions of other concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [7, 8, 18], and

recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [23]. As a result, many aspects of concurrent data objects have been researched deeply such as consistency conditions [1, 9, 20], concurrency hierarchy [6] and fault-tolerance [17].

In this paper, we look at another aspect of concurrent data objects: the freshness of the object states returned by read-operations. Freshness is a significant property for shared data in general and has achieved great concerns in databases [3, 12, 19] as well as in caching systems [13, 15, 16]. Briefly, freshness is a yardstick to evaluate how fresh/new a value of a concurrent object returned by a read-operation is, when the object is updated and read concurrently. For concurrent data objects, although read-operations are allowed to return any value written by other concurrent operations, they are preferred to return the freshest/latest one of these valid values, especially in reactive/detective systems. For instance, monitoring sensors continuously concurrently input data via a concurrent object and the processing unit periodically reads the data to make the system react accordingly. In such systems, the freshness of data influences how fast the system reacts to environment changes.

However, there are few results on the freshness problem in the literature. Simpson [21, 22] suggested a freshness specification for a single-writer-to-single-reader asynchronous communication mechanism, which is different from atomic register suggested by Lamport [14]. Simpson's communication model with a single writer and a single reader is not suitable for fully concurrent shared objects that many readers and many writers can concurrently access.

These issues motivate us to define and attack the freshness problem for wait-free shared objects. We model the problem as an online problem and then present two algorithms for it. The first one is a deterministic algorithm, which is a natural adaptation from an online search algorithm called *reservation price policy* [5]. The algorithm achieves a competitive ratio $\sqrt{\alpha}$, where α is a function of execution-time upper-bound of wait-free operations. Subsequently, we prove that the algorithm is optimal by proving that $\sqrt{\alpha}$ is the best competitive ratio for deterministic algorithms. The second is a new competitive randomized algorithm with competitive ratio $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$. The randomized algorithm is nearly optimal since our results [4] from an elaboration on the EXPO search algorithm [5] showed that $O(\ln \alpha)$ is an asymptotically optimal competitive ratio for randomized freshness algorithms.

The paper is organized as follows. Section 2 describes the freshness problem and models it as an online problem. Section 3 presents the optimal deterministic algorithm. Section 4 presents the randomized algorithm. The competitive ratio in this case is the expected value against an oblivious adversary. (We presume that the reader is familiar with competitive analysis of online algorithms, cf. [2].)

2 Problem and Model

Linearizability [9] is the correctness condition for concurrent objects. It requires that operations on the objects appear to take effect atomically at a point of time in their execution interval. This allows a read operation to return any of values written by concurrent write operations, which is illustrated by Figure 1.

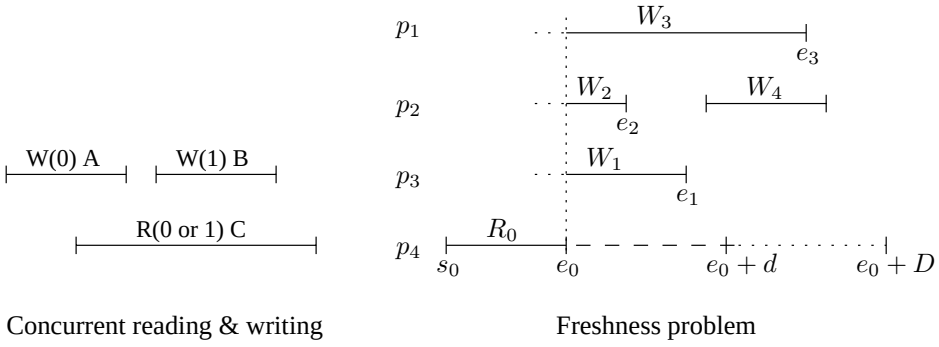

Fig. 1. Illustrations for concurrent reading/writing and freshness problem

We use "W(x) A" ("R(x) A") to stand for a write (read) operation of value x to (from) a shared register by process A. It is correct for C to return either 0 or 1 with respect to linearizability. However, from freshness point of view we prefer C to return 1, the newer/fresher value of the register. The freshness problem is to find a solution for read operations to obtain the freshest value from a shared object. Intuitively, if a read operation lengthens its execution interval by putting some delay between the invocation and the response, it can obtain a fresher value but it will respond more slowly from application point of view. Therefore, the freshness problem is to design read-operations that both respond fast and return fresh values.

The freshness problem is especially interesting in reactive systems, where monitoring sensors continuously and concurrently input data for a processing unit via a concurrent data object. The unit periodically reads the data from the object and subsequently makes the system react to environment changes accordingly. In order to react fast, the read-operation used by the unit must both respond fast and return a value as fresh as possible. If the read-operation responds immediately at time e_0 and an environment change occurs at time $e_0 + \epsilon$, the system must wait for a period T until the next read in order to observe the change. In this scenario, the system will react faster if the read-operation delays a bit to return the fresh value at $e_0 + \epsilon$. The system will subsequently react according to the change at time $e_0 + \epsilon$ instead of waiting until time $e_0 + T$ to be able to observe the change, where $\epsilon << T$ (Assume that processing time is negligible.).

The freshness problem is illustrated by Figure 1. In the illustration, a read operation R_0 runs concurrently to three write operations W_1, W_2 and W_3 on a concurrent shared object. In this paper, read/write operations imply operations on the same object. The actual execution interval of a operation i is defined from the time s_i the operation starts to the time e_i it takes effect (i.e. linearization point [9]). A time axis is from left to right. The value returned by R_0 becomes fresher if there are more end-points e_i appear in the interval $[s_0, e_0]$. In the illustration, if R_0 delays the time-point e_0 to $e'_0 = e_0 + d$, the execution interval $[s_0, e'_0]$ will include two more end-points e_1 and e_2 and thus the value returned is newer. However, the delay will also make the read-operation respond more slowly. This implies that R_0 needs to find the time delay d so as to maximize

the freshness value $f_d = \frac{k(|we_d|)}{h(d)}$, where $|we_d|$ is the number of new write-endpoints earned by delaying R_0's read-endpoint an interval d and k, h are increasing functions that depend on real applications. The k and h functions should be increasing in order to model progressive systems. Each application may specify its own functions k and h according to the relation between the latency and freshness in the application.

Assume that the shared object supports a function for read operations to check how many write operations (with their timestamp) are ongoing at a time[1]. A write-timestamp wt shows the *start-point* of the corresponding write operation whereas a read-timestamp rt shows the *end-point* of the corresponding read operation. The timestamp objective is to help R_0 ignore W_4 due to $rt_0 < wt_4$. Note that R_0 only needs to consider write-endpoints of write operations that occur concurrently to R_0 in its original execution interval $[s_0, e_0]$, e.g. R_0 will ignore W_4. Therefore, in the freshness problem, the number of concurrent write operations that have not finished at the original read-endpoint e_0 is known and is called M. This number is also the total number of considered write-endpoints, i.e. $|we_d| \leq M$.

The most challenging issue in the freshness problem is that the end-points of concurrent write operations appear unpredictably. In order to analyze the problem, we consider it as an online game between a player and an oblivious adversary where the malicious adversary decides when to place the write-endpoints e_i on-the-fly and the player (the read operation) decides when she should stop and place her read-endpoint e'_0. The online game starts at the original read-endpoint e_0 and the player knows the total number of write-endpoints M that the adversary will use throughout the game. At a time t, the player knows how many of M end-points have been used by the adversary so far, i.e. $|we_t|$, (by comparing M with the number of ongoing write operations that ran concurrently with the original read operation) and computes the current freshness value $f_t = \frac{k(|we_t|)}{h(t)}$. For each f_t observed, without knowledge of how the value will vary in the future, the player must decide whether she accepts this value and stops or waits for a better one. In this online game, the player's goal is to minimize the competitive ratio $c = \frac{f_{max}}{f_{chosen}}$, where f_{chosen} is the freshness value chosen by the player and f_{max} is the best value in this game, which is chosen by the adversary. The duration of this game D is the upper bound of execution time of the wait-free read/write operations and is known to the player. This implies that all the M write-endpoints must appear at a time-point in the interval, i.e. $|we_D| = M$.

In summary, we define the freshness problem as follows. Let M be the number of ongoing wait-free write operations at the original read-endpoint e_0 of a wait-free read operation and D be the execution-time upper-bound of these wait-free read/write operations. The read operation needs to find a delay $d \leq D$ for its new end-point e'_0 so as to achieve an optimal freshness value $f_d = \frac{k(|we_d|)}{h(d)}$, where $|we_d|$ is the number of write-endpoints earned by the delay d and k, h are increasing functions that reflect the relation between latency and freshness in real applications. The read-operation is only allowed to read the object data and check the number of ongoing write-operations. The write-operation is only allowed to write data to the object. We assume the time

[1] The assumption is practical since this can be done by adding a list of timestamps of ongoing write operations to the shared object.

is discrete, where a time unit is the period with which the read operation regularly checks the number of ongoing write operations on the shared object. The extended read operation is still wait-free with an execution-time upper-bound $2D$.

The rest of this paper presents two competitive online algorithms for the freshness problem. The first one is an optimal deterministic algorithm with competitive ratio $\sqrt{\alpha}$, where $\alpha = \frac{h(D)}{h(1)}$. The second one is a nearly-optimal randomized algorithm with competitive ratio $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$. Note that the competitive ratios do not depend on k and M, which are related to the number of end-points.

3 Optimal Deterministic Algorithm

Modeling the freshness problem as an online game, we observe that the freshness problem is a variant of online search [5]: In that problem, a player searches for the maximum (minimum) price in a sequence of prices that unfolds daily. For each day i, the player observes a price p_i and must decide whether to accept this price or to wait for a better one. The game ends when the player accepts a price, which is also the result.

Inspired by an online search algorithm called *reservation price policy* [5], we suggest a competitive deterministic algorithm for the freshness problem. In addition to the fact that the player is searching for the best in a sequence of freshness values that unfolds sequentially in a foreknown range, there are more restrictions on the adversary. Freshness values f_t at time t must fulfill:

$$\frac{f_{t-1} * h(t-1)}{h(t)} = \frac{k(|we_{t-1}|)}{h(t)} \leq f_t = \frac{k(|we_t|)}{h(t)} \leq \frac{k(M)}{h(t)} \tag{1}$$

The restrictions come from the fact that the adversary cannot remove the end-points she has placed, i.e. $|we_{t-1}| \leq |we_t| \leq M$, where $|we_t|$ is the number of end-points that have appeared until a time t, and the freshness value at the time t is $f_t = \frac{k(|we_t|)}{h(t)}$, where k, h are increasing functions. The restrictions make the adversary in the freshness problem weaker than the adversary in the online search problem, and intuitively the player in the freshness problem should benefit from this. However, we will prove that this is not the case for deterministic algorithms (cf. Theorem 2).

Before presenting the deterministic freshness algorithm, we need to find upper/lower bounds on freshness values f_t. Since $1 \leq t \leq D$, from Equation (1) it follows $f_t \leq \frac{k(M)}{h(1)}$. On the other hand, since M ongoing write-operations must end at time-points in the interval D, the player is ensured a freshness value $f_{min} = \frac{k(M)}{h(D)}$ by just waiting until $t = D$. Therefore, the player considers to stop at a freshness value f_t only if $f_t \geq \frac{k(M)}{h(D)}$. We have $\frac{k(M)}{h(D)} \leq f_t \leq \frac{k(M)}{h(1)}$.

Deterministic Algorithm: The read operation accepts the first freshness value that is not smaller than $f^* = \frac{k(M)}{\sqrt{h(1)h(D)}}$.

Indeed, let f^* be the threshold for accepting a freshness value and f_{max} be the highest value chosen by the adversary. The player (the read operation) waits for a value $f_t \geq f^*$. If such a value appears in the interval D, the player accepts it and returns it as

the result. Otherwise, when waiting until the time D, the player must accept the value $f_{min} = \frac{k(M)}{h(D)}$.

Case 1: If the player chooses a big value as f^*, the adversary will choose $f_{max} < f^*$, causing the player to wait until the time D and accept the value $f_{min} = \frac{k(M)}{h(D)}$. The competitive ratio in this case is $c_1 = \frac{f_{max}}{\frac{k(M)}{h(D)}} < \frac{f^*}{\frac{k(M)}{h(D)}}$.

Case 2: If the player chooses a small value as f^*, the adversary will place f^* at a time t, causing the player to accept the value and stop. Right after that, the adversary places all M end-points, achieving a value $f_{max} = \frac{k(M)}{h(t)} \leq \frac{k(M)}{h(1)}$ (equality occurs when the adversary chooses $t = 1$). The competitive ratio in this case is $c_2 = \frac{\frac{k(M)}{h(1)}}{f^*}$.

The player chooses f^* so as to make $c_1 = c_2$, which results in $f^* = \frac{k(M)}{\sqrt{h(1)h(D)}}$ and the competitive ratio $c = c_1 = c_2 = \sqrt{\frac{h(D)}{h(1)}}$. This leads to the following theorem.

Theorem 1. *The suggested deterministic algorithm is competitive with competitive ratio $c = \sqrt{\alpha}$, where $\alpha = \frac{h(D)}{h(1)}$.*

We now prove that no deterministic algorithm can do better.

We use a logarithmic vertical axis for freshness. Let LF denote the logarithm of freshness. More specifically, we normalize the LF axis so that freshness $\frac{k(M)}{h(D)}$ corresponds to point 0 and freshness $\frac{k(M)}{h(1)}$ corresponds to point $\ln \frac{h(D)}{h(1)} = \ln \alpha$. One unit on the LF axis multiplies the freshness by factor e (Euler's number).

We also introduce some parameters that characterize the status of a game. Let t be the time, initially $t = 1$. At any moment, let f be the maximum LF the adversary has already reached during the history of the game, and g the maximum LF the adversary can still achieve at a given time. LF value $g(t)$ at time t corresponds to freshness $k(M)/h(t)$, unless f is already larger, in which case we have $g = f$. However in the latter case the game is over, without loss of generality: The adversary cannot gain more and would therefore decrease the freshness as quickly as possible, in order to make the player's position as bad as possible, hence an optimal player would stop now. (The dotted polyline in Figure 2 illustrates the case $f = g(t)$ in which the player should stop at time t.)

The horizontal axis is for the logarithm of $h(t)$. We normalize it so that $h(1)$ corresponds to point 0 and $h(D)$ corresponds to point $\ln \frac{h(D)}{h(1)} = \ln \alpha$). Note that, in these logarithmic coordinates, g simply decreases at unit speed, starting at point $\ln \alpha$. Finally, let c denote the current LF. We remark that c can decrease at most at unit speed but can jump upwards arbitrarily as long as $c \leq g$.

Theorem 2. *The optimal deterministic competitive ratio is asymptotically (subject to lower-order terms) $\sqrt{\alpha}$, where $\alpha = \frac{h(D)}{h(1)}$.*

Proof. We only need to show an adversary strategy that enforces the claimed competitive ratio. Our logarithmic coordinates make the argument rather simple: The adversary

starts with $c = \frac{\ln \alpha}{2} = \frac{\ln \frac{h(D)}{h(1)}}{2}$. Then she decreases c at unit speed until the player stops. Immediately after this moment, c jumps to g if $c > 0$ at the stop time (Case 1), otherwise c keeps on decreasing at unit speed (Case 2). Clearly, we have constantly $g - c = \frac{\ln \alpha}{2}$ until the stop time. Let p be the player's value of LF. In Case (1) we finally get $f = g$, hence $f - p = g - c = \frac{\ln \alpha}{2}$ (cf. the dashed polyline c_1 in Figure 2). In Case (2), f has still its initial value $\frac{\ln \alpha}{2}$ whereas $p \leq 0$, hence $f - p \geq \frac{\ln \alpha}{2}$ (cf. the line c_2 in Figure 2). Thus the competitive ratio is at least $e^{\frac{\ln \alpha}{2}} = \sqrt{\alpha}$. □

We have shown that a deterministic player cannot benefit from the constraints on the behaviour of freshness in time (compared to the unrestricted online search problem).

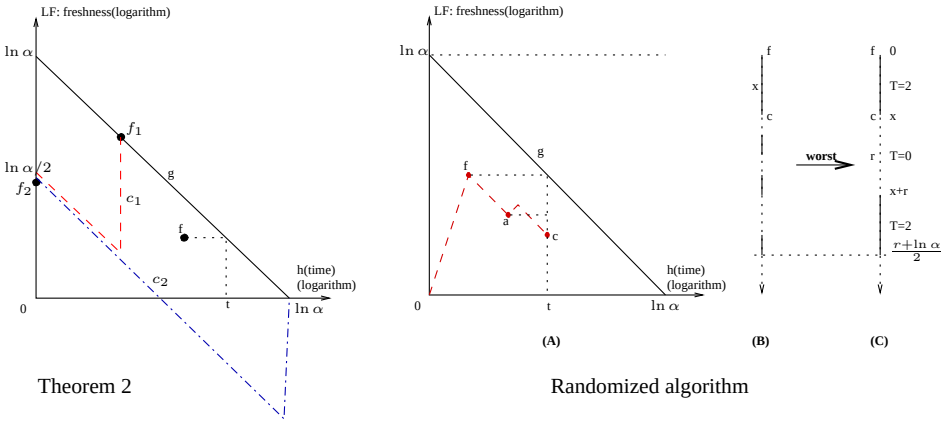

Fig. 2. Illustrations for Theorem 2 and the randomized algorithm

4 Competitive Randomized Algorithm

Next we present a randomized algorithm for the freshness problem, against the oblivious adversary [2]. It achieves a competitive ratio $c = \frac{\ln \alpha}{1 + \ln 2 - \frac{2}{\sqrt{\alpha}}}$, where $\alpha = \frac{h(D)}{h(1)}$.

As discussed in the previous section, our problem is a restricted case of online search. We model the problem by a game between an (online) player and an adversary. The adversary's profit is the highest freshness ever reached. The player's profit is the freshness value at the moment when she stops. Note that for a player running a randomized strategy, the profit is the expected freshness value, with respect to the distribution of stops resulting from the strategy and input. We shall make use of a known simple transformation of (randomized) online search to (deterministic) one-way trading [5]: The player has some budget of money she wants to exchange while the exchange rates may vary over time. Her goal is to maximize her gain. The transformation is given as follows: The budget corresponds to probability 1, and exchanging some fraction of money means to stop the game with exactly that probability. Note that a deterministic

algorithm for online search has to exchange all money at *one* point in time. For the freshness problem, it is possible to apply a well-known competitive randomized algorithm EXPO [5]. Applying the EXPO algorithm on the freshness problem achieves a competitive ratio $\ell \frac{2^{\ell-1+1/\ln 2}}{2^{\ell-1+1/\ln 2}-\frac{1}{\ln 2}}$, where $\ell = \log_2 \alpha$. That means for the freshness problem our randomized algorithm is better than the EXPO algorithm by a constant factor $\frac{1+\ln 2}{\ln 2}$ when α becomes large.

Theorem 3. *There is a randomized algorithm for the freshness problem with expected competitive ratio* $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$ *against an oblivious adversary, where* $\alpha = \frac{h(D)}{h(1)}$.

Proof. We start with some conventions. We imagine that the money, both exchanged and non-exchanged, is "distributed" on the LF axis. Formally, the allocation of money on the LF axis at any time is described by two non-negative real *density functions* S and T, where $S(x)$ is the density of not yet exchanged money in point x of the LF axis, $T(x)$ is similarly defined for the money that has been already exchanged. What functions S and T specifically are, and how they are modified by the opponents' actions, will be described below. Let the total amount of money be $\ln \alpha$ by convention. (Recall that scaling factors do not influence the competitive ratio.)

The *value* of every piece of *exchanged* money is the freshness value of its position on the LF axis. Note that the total value of exchanged money defined in this way, i.e. the integral over the value-by-density product, is the player's profit in the game. Moreover, the player can temporarily have some of the money in her *pocket*.

The idea of the strategy is to guarantee some concentration of exchanged money immediately below the final f, either some constant minimum density of T or, even better, a constant amount at one point not too far from f. We want to keep T simple in order to make the calculations simple. (The well-known δ_x symbol used below denotes the distribution with infinite density at a single point x but with integral 1 on any interval that contains x. We also use the same notations f, g, c as earlier.) Locating much money instantaneously is risky because c may jump upwards, and then this money has little value compared to the adversary's. On the other hand, since c decreases at most with unit speed, the player may completely abstain from exchanging money as long as c is increasing, and wait until c goes down again. These preliminary thoughts lead to the following strategy.

In the beginning, let the not-yet-exchanged money be located on the LF axis on interval $[0, \ln \alpha]$ with density 1, that is, we have $S = 1$ on this interval. Remember that g decreases at unit speed. The player puts the money above g in her pocket. Whenever f increases, she also puts the money below the new f in her pocket. Hence we always have $S = 1$ on $[f, g]$, and $S = 0$ outside. The player continuously locates exchanged money on the LF axis, observing the following rule: *If you have money in your pocket and c is positive and decreasing, and $T(c) < 2$ at the current c, then set $T(c) := 2$. If the game is over (because of $f = g$) and not all money is exchanged yet, put the rest r on the current c.* Note that the adversary must set the final c nonnegative.

Filling-up density T to 2 is always possible: The player uses the one unit of money from S that she gets per time unit from the region above the falling g, and the money from S that she got directly from the current points c when f went upwards.

Obviously, the player produces a density function T that is constantly 2 on certain intervals and 0 outside, plus some component $r\delta_c$. We make some crucial observations regarding the final situation: (1) T has density 2 on interval $(c, f]$, or we have $c = f$. (2) The *gaps* with $T = 0$ between the "$T = 2$ intervals" have total length at most r.

These claims follow easily from the strategy: (1) Either c begins decreasing, starting from the last f, and T is filled up to 2 all the time when $c > 0$, as we saw above, or the final c equals the final f. (2) Whenever f went upwards, the player has taken from S the money corresponding to the increase of f, and later she has transferred it to T and located it at the same points again. Hence, only on intervals not "visited" again by c we have $T = 0$, and the money taken from S on these intervals is still in the player's pocket and thus contributes to r.

Figure 2-(A) illustrates the player's behavior. The dashed line represents a variation of c in a game; point c is the final value of c when the game ends, i.e. $f = g(t)$. For all values v on the LF axis between f and a and between a and c, the player sets $T(v) = 2$.

Using (1),(2) we now analyze the profit the player can guarantee herself. Remember that the value of exchanged money located on the LF axis decreases exponentially. Let $x = f - c$ (final values). Both r and x depend on the input, i.e., the behavior of c in time. The total amount of money is fixed, it equals $\ln \alpha$. For any fixed r, x, the worst case is now that the gaps in T sum up to the maximum length r and are as high as possible on the LF axis, that is, immediately below point c, because in this case all exchanged money outside $[c, f]$ has the least possible value. That is, T has only one gap, namely interval $[c - r, c]$.

Figure 2-(C) illustrates the worst case corresponding to an instance -(B), where solid lines represent ranges on the LF axis with $T = 2$. In the worst case, the adversary shifts all solid lines except for $[c, f]$ to the lowest possible position so as to minimize the player's profit.

Hence, a lower bound on the player's profit, divided by the value at f, is given by

$$\min_{r,x} \left(2 \int_0^x e^{-t} dt + r e^{-x} + 2 \int_{x+r}^{(r+\ln \alpha)/2} e^{-t} dt \right),$$

where we started integration (with $t = 0$) at point f and go down the LF axis (cf. Figure 2-(C)). Verify that, in fact, $\int T dt = \ln \alpha$. The above expression evaluates to

$$2 + (r - 2 + 2e^{-r})e^{-x} - 2e^{-(r+\ln \alpha)/2} > 2 + (r - 2 + 2e^{-r})e^{-x} - 2/\sqrt{\alpha}.$$

For any fixed x, this is minimized if $2e^{-r} = 1$, that is, $r = \ln 2$. Since now $r - 2 + 2e^{-r} = \ln 2 - 2 + 1 < 0$, the worst case is $x = 0$, which gives $1 + \ln 2 - 2/\sqrt{\alpha}$. The adversary earns $\ln \alpha$ times the value at f. □

References

1. Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. ACM Trans. Comput. Syst. **12**(2) (1994) 91–122
2. Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press (1998)

3. Cho, J., Garcia-Molina, H.: Synchronizing a database to improve freshness. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data. (2000) 117–128
4. Damaschke, P., Ha, P.H., Tsigas, P.: One-way trading with time-varying exchange rate bounds. Technical report CS:2005-17, Chalmers University of Technology (2005)
5. El-Yaniv, R., Fiat, A., Karp, R.M., Turpin, G.: Optimal search and one-way trading online algorithms. Algorithmica **30**(1) (2001) 101–139
6. Gafni, E., Merritt, M., Taubenfeld, G.: The concurrency hierarchy, and algorithms for unbounded concurrency. In: Proc. of Symp. on Principles of Distributed Computing (PODC). (2001) 161–169
7. Harris, T.: A pragmatic implementation of non-blocking linked lists. In: Proc. of the Intl. Symp. on Distributed Computing (DISC). (2001) 300–314
8. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA). (2004) 206–215
9. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. on Programming Languages and Systems **12**(3) (1990) 463–492
10. Herlihy, M.: Wait-free synchronization. ACM Trans. on Programming and Systems **11**(1) (1991) 124–149
11. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Trans. on Programming Languages and Systems **15**(5) (1993) 745–770
12. Kang, K.D., Son, S.H., Stankovic, J.A.: Managing deadline miss ratio and sensor data freshness in real-time databases. IEEE Trans. on Knowledge and Data Engineering **16**(10) (2004) 1200–1216
13. Labrinidis, A., Roussopoulos, N.: Exploring the tradeoff between performance and data freshness in database-driven web servers. The VLDB Journal **13**(3) (2004) 240–255
14. Lamport, L.: On interprocess communication. part ii: Algorithms. Distributed Computing **1**(2) (1986) 86–101
15. Li, W.S., Po, O., Hsiung, W.P., Candan, K.S., Agrawal, D.: Engineering and hosting adaptive freshness-sensitive web applications on data centers. In: Proc. of the Intl. Conf. on World Wide Web. (2003) 587–598
16. Ling, Y., Chen, W.: Measuring cache freshness by additive age. SIGOPS Oper. Syst. Rev. **38**(3) (2004) 12–17
17. Malkhi, D., Merritt, M., Reiter, M.K., Taubenfeld, G.: Objects shared by byzantine processes. Distrib. Comput. **16**(1) (2003) 37–48
18. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. of Symp. on Principles of Distributed Computing (PODC). (1996) 267–275
19. Pacitti, E., Simon, E.: Update propagation strategies to improve freshness in lazy master replicated databases. The VLDB Journal **8**(3-4) (2000) 305–318
20. Shao, C., Pierce, E., Welch, J.L.: Multi-writer consistency conditions for shared memory objects. In: Proc. of the Intl. Symp. on Distributed Computing (DISC). (2003) 106–120
21. Simpson, H.R.: Correctness analysis for class of asynchronous communication mechanisms. Computers and Digital Techniques, IEE Proc.- **139**(1) (1992) 35– 49
22. Simpson, H.R.: Freshness specification for a class of asynchronous communication mechanisms. Computers and Digital Techniques, IEE Proc.- **151**(2) (2004) 110–118
23. Sundell, H., Tsigas, P.: NOBLE: A non-blocking inter-process communication library. In: Proc. of the Workshop on Languages, Compilers and Run-time Systems for Scalable Computers. LNCS, Springer Verlag (2002)