Rewriting “The Turtle and the Hare”: Sleeping to Get There Faster[*]

José Pereira
jop@di.uminho.pt
U. Minho

Rui Oliveira
rco@di.uminho.pt
U. Minho

1. Introduction

When developing algorithms for dependable distributed systems one often makes several simplifying assumptions that are essential to reason about the problem in hand. It is usual to assume an asynchronous system model, unconstrained system resources and the absence of easily maskable faults such as message loss. While most of the simplifications strengthen the model and are particularly useful when proving theoretical edge results, asynchrony, on the contrary, is a "non-assumption" and it is specially appealing in practice as it yields robust solutions that are correct regardless of the actual timing behavior of the target systems.

The asynchronous model is useful even when the problem cannot be solved in a pure asynchronous system (e.g. consensus [2]) or the specification itself includes timeliness requirements. By factoring out timeliness assumptions one can separately reason about safety guarantees [13] or constrain assumptions on time to a small subset of the entire system, therefore increasing coverage and simplifying implementation [20].

In an asynchronous model the performance of distributed algorithms is typically evaluated based on theoretical metrics such as the message complexity or communication steps. Such metrics are however of limited use at predicting actual performance as understood by systems engineers and the best measurements are obtained with trade-offs between different theoretical metrics [9]. The systems approach relies heavily on actual timing measurements and resource usage of real systems or very realistic, although informal, models. A good example is the usage of large benchmarking infrastructure such as PlanetLab [16] and Netbed [22] or detailed simulators such as ns-2 [11].

Moreover, testing in real systems shows that the optimum performance in each system is obtained with a different algorithmic trade-off. Changing the trade-off however, means changing the original algorithm which was previously designed to exclude all runs but those optimal in a given theoretical metric. This is obviously a case where premature optimization is the root of some evil, since having to change the algorithm negates the advantages of separately obtaining correctness proofs.

In this paper we advocate that instead of optimizing for a single metric, the goal of the computer scientist when designing algorithms in an asynchronous system model should be to:

- Accommodate a wide range of runs that fare differently with various theoretical metrics. This is contrary to the common approach in which all runs but those that are optimal in the selected metric are excluded.
- Ensure that the likelihood of different runs can be manipulated by the implementation without impact in correctness.

The resulting algorithms thus give additional freedom to the implementor to optimally tune the implementation for each target system, unconstrained by premature performance assumptions.

The cornerstone of the approach is that configuration for performance is achieved solely by *introducing finite delays* in the implementation. By manipulating only time, the correctness of algorithms proven in an asynchronous model is kept ensured.

We illustrate the approach with two examples. First, in Section 2 we describe an atomic broadcast with optimistic delivery [19] that makes a very simple use of the proposed approach to select the runs in which the predicted order most closely matches the final order in each system. The second example, Section 3, describes a consensus protocol that uses the proposed approach to radically mutate its message exchange pattern [15]. Besides mimicking existing protocols, it can be configured to use an innovative gossip-style pattern and thus scale gracefully to very large systems.

The paper concludes with a brief review of related work in Section 4 and a discussion of the key requirements and consequences of the approach in Section 5.

2. Case study: Optimistic total order in WAN

An atomic broadcast protocol delivers messages by an agreed total order, thereby easing the implementation of

[*] Supported by FCT, project RUMOR (POSI/CHS/40088/2001).

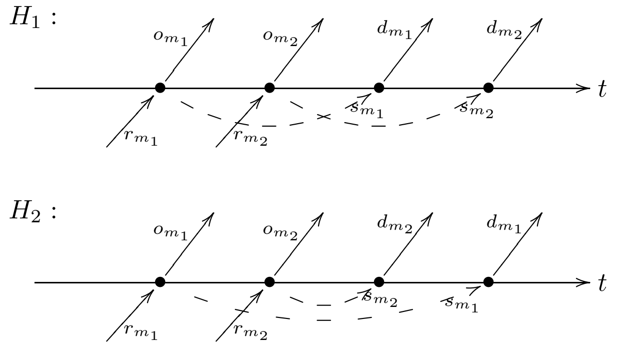
fault-tolerant services using the replicated state machine approach [18]. The requirement of agreement introduces a significant latency overhead when compared with a reliable broadcast protocol. Additional latency is introduced if the protocol further provides uniform agreement [21].

It has been pointed out that in some total ordering protocols, such as those based on consensus or on a sequencer [3], the total order decided is the spontaneous ordering of messages as observed by one of the participating processes. In addition, in local area networks (LANs) it can be observed that the spontaneous order of messages is often the same in all processes, which therefore are able to accurately predict the agreed order.

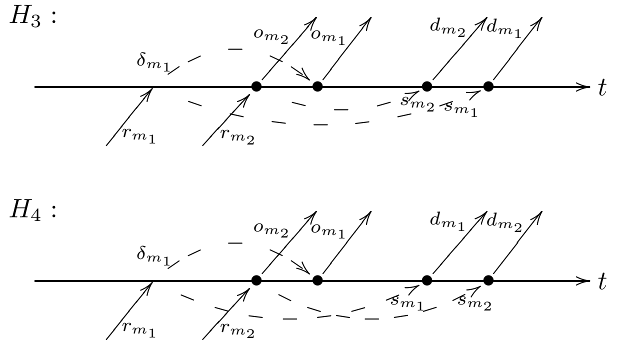
This observation can be useful to mask (although not reduce) delivery latency, by tentatively delivering messages based on spontaneous ordering, thus allowing the application to proceed the computation in parallel with the ordering protocol [10]. Later, when the total order is established and if it confirms the optimistic ordering, the application can immediately use the results of the optimistic computation. If not, it must undo the effects of the computation, *e.g.* by aborting a transaction, and restart it using the correct ordering. The effectiveness of the technique rests on the assumption that a large share of correctly ordered tentative deliveries offsets the cost of undoing the effects of mistakes.

Premature optimization A protocol exploiting optimistic delivery [10] is quite simple and can be obtained from the original ordering protocol by adding the following: upon reception, *immediately* signal optimistic delivery thus determining the optimistic ordering. This produces runs where optimistic deliver happens *atomically* with reception. Considering that r_m, o_m, s_m, d_m denote, respectively, reception, optimistic delivery, sequencing and final delivery of a message m, this produces runs such as H_1 and H_2 in Figure 1(a). In detail, optimistic order correctly predicts total order in H_1 but misses in H_2, requiring the effects of the computation based on optimistic deliveries to be undone.

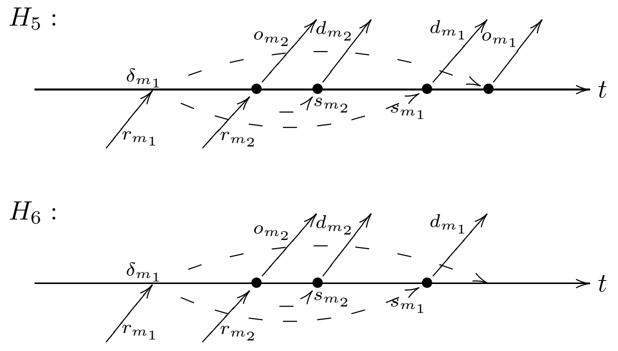
This protocol cannot however produce runs H_3 and H_4 presented in Figure 1(b), because optimistic delivery would have to be executed atomically upon reception. This is unfortunate, as H_3 would have saved a rollback of the optimistic processing of message m_1. An example of why runs such as H_3 are useful is the common loopback optimization in a protocol stack that leads the sender to observe its own messages sooner than everyone else. In an optimistic atomic broadcast protocol, this would mean that a sender would always miss on the ordering of its own messages. If allowed by the specification, an implementor would most certainly find a way to compensate the loopback optimization.



(a) Spontaneous ordering.



(b) Manipulating optmistic order with delays.



(c) Other runs allowed by order manipulation.

Figure 1. Runs with optimistic delivery.

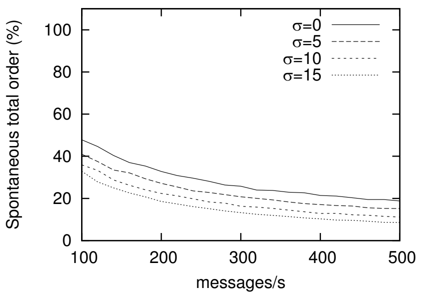
Algorithm A better algorithm can thus be obtained by decoupling the assumption that the agreed order of messages can be predicted from how it can be predicted. Such protocol, is obtained as follows: upon reception, schedule the optimistic delivery of each message m after a finite and possibly zero delay δ_m, thus allowing some yet unknown optimistic ordering criterion to be implemented later, thus allowing H_3. Although seemingly trivial, this modification has to be anticipated at the algorithmic level and is not a simple implementation decision. Consider runs H_5 and H_6 in Figure 1(c), which are also allowed after removing the atomicity requirement on reception and optimistic delivery. Although run H_5 should be obviously excluded, run H_6 is probably acceptable. The decision to exclude or include either would have an impact on the specification of the protocol and thus on the correctness proofs of both the protocol and the application.

At first sight, our new protocol is much worse performance-wise since we are trained to regard delays in the asynchronous system model as random, or worse, controlled by an evil adversary. Indeed, the change has apparently opened up the possibility for the adversary to mess with the optimistic ordering by decoupling it from spontaneous order.

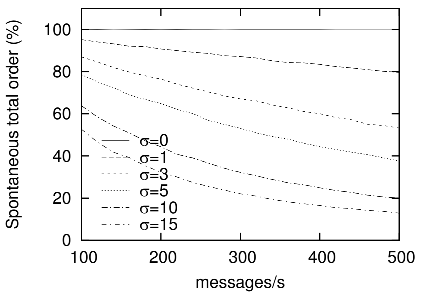
Implementation The actual performance of the algorithm in a real system depends essentially on a high probability of good runs and not on the total absence of bad runs. It is therefore up to the implementation to ensure with a high probability that the delays are judiciously chosen to obtain a good match with final ordering. Besides coping with the loopback optimization, the idea can be extended to cope with the variability of end-to-end delays in a wide-area network: Even if message transmission delays between any given pair of processes is stable, each process will observe first and thus optimistically deliver messages from those that are closer in terms of network delays, thus missing spontaneous ordering of concurrent messages.

An adaptive protocol to optimize delays is presented in [19] and works as follows. When issuing a sequence number, the sequencer process tags it with the locally measured elapsed time since the previous reception. Other participants, are therefore able to compare such duration with the same interval measured locally and thus adjust delays that are imposed on messages from the same sources. Whenever possible, the adjustment is done by reducing the currently estimated delay on the message that turned out to be late. If not possible, because no additional delay is being used, the delay on the other source is increased.

When the network is stable (i.e. standard deviation σ is low), delays imposed by each process to messages from each source converge to a fixed value as shown by detailed simulation and experiments with a prototype [19]. The result is an improvement on the predicted optimistic order when compared to spontaneous ordering on reception, as shown in Figure 2, while at the same time optimistic deliver precedes final delivery by enough time to be useful for optimistic processing.



(a) Spontaneous ordering.



(b) Optimistic ordering.

Figure 2. Comparison of spontaneous with optimistic order after delay compensation.

Detailed presentation of the protocol, the implementation, the evaluation environment, and further results can be found in [19].

3. Case study: Mutable consensus

Several distributed programming problems such as atomic broadcast, view synchrony and (weak) atomic commitment can be reduced to consensus. We focus on protocols based on unreliable failure detectors [2, 17] to circumvent the FLP impossibility result [4] in asynchronous message passing systems when processes may fail by crashing. In particular, protocols using an eventually strong failure detector and that assume a majority of non-faulty processes. These protocols execute in asynchronous rounds with a rotating coordinator. In each round, an estimate is broadcast to all participants by the coordinator of the round. Processes then try to gather a majority of votes, to decide on the coordinator's estimate or to proceed to the next round. When a value is decided it is reliably broadcast to all participants. These protocols differ mostly on the message exchange pattern used to collect votes and disseminate the decision, which has a definite impact in performance [1].

Premature optimization In a centralized protocol [2], all votes are gathered by the round's coordinator. In detail, when entering a round the coordinator collects estimates from other participants. Then it broadcasts a selected estimate and collects the acknowledgments. Upon receiving acknowledgments from a majority of the participants, the decision is broadcast. This allows the decision to be reached in three communication steps and requires that only the coordinator handles messages from all participants.

On the other hand, in a decentralized protocol [17] all votes are broadcast to all participants, making it possible that each process independently gathers a majority and thus reaches a decision. This allows the decision to be reached in two communication steps at the expense of a larger number of messages exchanged. Network bandwidth can be reduced by using broadcast mechanisms at the network level when available, but still requires that all participants handle messages from all others.

Notice that, in terms of communication, the correctness of both protocols depends only on the guarantee that a majority of votes is relayed to all participants. A way to achieve this is to have, in each round, a single trusted process to gather all the votes and afterwards relay just the decision to all other processes. The problem however, is that existing protocols follow this approach and absolutely preclude runs in which the relaying of the votes themselves is done by more than one process, even when there are no faults. As an example, it could be interesting to collect votes using a tree structure to reduce the number of messages sent and received by each process.

This is unfortunate as, in practice, the performance of a distributed protocol in general, and in particular its scalability to large numbers of participants, is tightly related to the number of messages sent and received by each process. The available processing power of such participants thus directly translates to an upper bound on the scalability of the protocols.

Algorithm We start with a decentralized consensus protocol based on an unreliable channel abstraction [12] and do the following modification: every process immediately relays votes as it receives them, instead of waiting for a majority to broadcast the decision. Notice that this simple modification is sufficient to allow runs in which a decision is reached by the first time by a process without having received direct communication from a majority of participants.

At first sight, this protocol is even worse than the original, as we are trained by the usual assumption of reliable channels to consider that all messages consume valuable system resources and thus the proposed change results in a lot of redundant transmissions. Actually, it is not true that all messages sent have to be actually transmitted as the proposed protocol rests on stubborn channels [5]. Informally, a stubborn channel guarantees the delivery of only the last message sent. In practice, this can be implemented simply by buffering and periodically retransmitting the last message sent.

As messages can be lost by stubborn channels, it is possible that only a small fraction of the messages sent by the protocol are actually transmitted through the underlying network. We can easily fabricate valid runs which exchange a much lower number of messages at the network level. It is highly unlikely that a naive implementation produces such desirable runs, though.

Implementation We therefore seek an implementation that maximizes the likelihood of desirable runs. Interestingly, this can be achieved simply by introducing finite delays in a naive implementation of stubborn channels, which makes it likely that a message becomes obsolete and is discarded before having been transmitted. Moreover, as delays are finite this does not in any way compromise the correctness of the protocol, which assumes an asynchronous system model. In practice, judiciously chosen delays make it likely that only desirable runs occur thus resulting in very good performance in practical metrics such as the latency, number of messages and the number of bytes transmitted.

Such delays avoid the actual transmission of a message m due to two different reasons. The first is that they increase the likelihood of a more recent message being sent in the meantime, which with stubborn channels discards all previously sent messages. The second is that if a decision can be reached by all processes before the delay expires, the transmission of m can be entirely avoided in practice. As an example, consider the usage of consensus to implement view synchronous multicast, in which an instance of consensus is run to install each view [7]. As soon as a process has started receiving messages (or acknowledgments to messages) from all others in the recently installed view, it knows that all participants in the previous consensus instance have decided. It may therefore terminate the consensus protocol and flush all pending messages.

Different configurations of delays lead to different messages being actually transmitted and thus result in different classes of desirable runs. Some of these resemble the message exchange pattern of well known protocols. Others result in innovative message exchange patterns with desirable performance characteristics.

A first intuition on the impact of the delays introduced in each mutation can be obtained from Figure 3, which

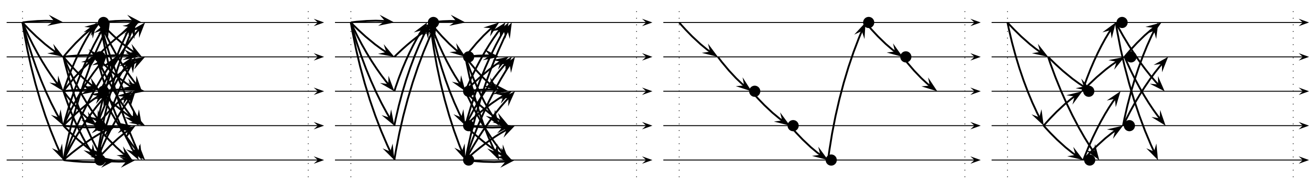

(a) Early (b) Centralized (c) Ring (d) Gossip

Figure 3. Prefixes of typical executions.

presents the graphical representation of prefixes of actual runs of a prototype of the mutable consensus protocol when combined with each implementation of the stubborn channels. In these pictures, arrows denote messages and solid dots the decision. The x-axis represents real-time. The entire duration of the interval presented is 1 ms. All messages actually transmitted during the interval are presented.

The first two of them mimic well known centralized and decentralized protocols [2, 17]. A third is called the *ring* and uses very little resources at the expense of high latency. Finally, the *permutation gossip* mutation provides a gossip-style message exchange pattern for consensus. Gossip-based protocols are used for a variety of distributed programming problems and are known for their scalability and resilience to network omissions.

Notice also that as correctness is not affected by the strategy used to compute the finite delays introduced, it is also not affected if each participant uses a different strategy, or even if a participant changes the strategy dynamically. This suggests that a mutable protocol can be an ideal foundation for the adaptation to a changing environment by requiring just the definition of a dynamic policy to compute delays.

A detailed presentation of the protocol, the implementation, the evaluation environment, and further results can be found in [15].

4. Related work

There is a plethora of work on the usage of time in an asynchronous system model. Such work is targeted at separating the guarantees of (timeless) safety and liveness from guarantees on timeliness or on providing encapsulated oracles that overcome the limitations of the asynchronous model [20, 13]. Neither propose the instrumental usage of time in the implementation stage as a mechanism to configure an algorithm for optimal performance by means of avoiding undesirable runs.

There are several proposals of total ordering protocols that take advantage of spontaneous ordering. The optimistic atomic broadcast [14] has no optimistic delivery but takes advantage of spontaneous ordering to optimize on the communication step metric. Protocols with optimistic delivery are useful to mask the latency of uniform agreement [21] and in the context of transaction processing where rollback is possible [10].

There are also several other proposals of configurable consensus protocols. Namely, the versatile consensus protocol [8] generalizes both proposals analyzed [2, 17] such that a predetermined subset of processes gather votes. There is also a proposal for a protocol that can use different oracles to overcome the FLP impossibility result [6]. Nevertheless, all of them share the limitation that at least one process relays messages from all others and cannot be reconfigured independently.

5. Discussion

This paper underlines the mismatch between theoretical performance metrics and actual systems performance, and proposes an approach to overcome it. In short, we propose algorithms in which the likelihood of runs with different performance trade-offs can be manipulated at the implementation level simply by introducing finite delays. This allows the implementation to be optimized in a concrete environment without endangering correctness which has been established on an asynchronous system model.

The approach is made possible by:

- Minimizing the assumptions on the system model, which includes using the asynchronous system model but also avoiding simplifications traditionally considered harmless such as reliable channels. Stubborn channels [5] provide an excellent trade-off.
- Avoiding premature optimization that overly restricts the runs possible to a subset that is optimal according to a theoretical metric, but which embodies false performance assumptions. By having concrete information about the target system, the implementor can do a better job at optimizing performance.

The approach is illustrated by two case-studies summarizing previous research results [19, 15]. The optimistic total order protocol shows how to dynamically compute delays to adapt to the environment and improve optimistic ordering. The mutable consensus protocol provides several static configurations that result in innovative message exchange patterns and, to the best of our knowledge, the only consensus protocol that scales to a very large number of participants by using gossiping.

An interesting consequence is that in both cases the computation of delays that reconfigure protocols can be done independently by participants without impact on correctness. This suggests a robust foundation for a configurable and adaptive distributed systems framework.

In the original tale, the perseverance of the turtle wins as the overly confident hare stops to sleep. Design and implementation of dependable distributed systems usually follows the approach of the turtle: First the computer scientist ensures that the turtle safely and eventually reaches the finish line. Then the implementor makes the heavy beast walk as fast as it can. The morale of this paper is that in dependable distributed systems taking some time to sleep might be the easiest way to win the race.

References

[1] O. Bakr and I. Keidar. Evaluating the running time of a communication round over the internet. In *ACM Symp. Principles of Distributed Computing*, July 2002.

[2] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), Mar. 1996.

[3] X. Défago, A. Schiper, and P. Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Technical Report DSC/2000/036, École Polytechnique Fédérale de Lausanne, Switzerland, Sept. 2000.

[4] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, Apr. 1985.

[5] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical Report 98-278, EPFLausanne, June 1998.

[6] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Trans. Computers*, 2003.

[7] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. Software Engineering*, 27(1), Jan. 2001.

[8] M. Hurfin, A. Mostefaoui, and M. Raynal. A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors. *IEEE Trans. Computers*, 51(4), Apr. 2002.

[9] I. Keidar. Challenges in evaluating distributed algorithms. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*. Springer, 2003.

[10] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Intl. Conference on Distributed Computing Systems*, June 1999.

[11] Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[12] R. Oliveira. *Solving consensus: From fair-lossy channels to crash-recovery of processes*. PhD thesis, EPFLausanne, Feb. 2000.

[13] R. Oliveira, J. Pereira, and A. Schiper. Primary-backup replication: From a time-free protocol to a time-based implementation. In *IEEE Intl. Symp. Reliable Distributed Systems*, Oct. 2001.

[14] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Intl. Symposium on Distributed Computing (DISC)*, Sept. 1998.

[15] J. Pereira and R. Oliveira. The mutable consensus protocol. In *IEEE Intl. Symp. Reliable Distributed Systems*, Oct. 2004.

[16] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. Planet-Lab: A blueprint for introducing disruptive technology into the Internet. In *ACM Workshop on Hot Topics in Networks*, Oct. 2002.

[17] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3), Apr. 1997.

[18] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*. Addison Wesley, 1993.

[19] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *IEEE Intl. Symp. Reliable Distributed Systems*, Oct. 2002.

[20] P. Veríssimo and A. Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers - Special Issue on Asynchronous Real-Time Systems*, Aug. 2002.

[21] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *IEEE Intl. Symp. Reliable Distributed Systems*, Oct. 2002.

[22] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *ACM Symposium on Operating System Design and Implementation*, Dec. 2002.