Membership-Oblivious Protocols

Fernando Pedone* Pascal Felber† Stefan Pleisch‡

*Università della Svizzera Italiana (USI), CH-6904 Lugano, Switzerland
E-mail: fernando.pedone@unisi.ch

†Institut Eurecom, 2229 route des Crêtes, B.P.193, 06904 Sophia Antipolis, France
E-mail: pascal.felber@eurecom.fr

‡École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
E-mail: stefan.pleisch@epfl.ch

Abstract

Many distributed protocols explicitly depend on membership information, such as the identity or number of sites that are currently part of the system. Given this importance, several mechanisms have been proposed over the past decades to handle membership information, but most of them do not scale well in large- and highly-dynamic systems. In this paper we advocate the use of "membership-oblivious" protocols, i.e., protocols that do not depend on the number of processes in the system, and therefore do not depend on membership. Our approach is based on the use of currency.

1. Introduction

A lot of effort has been put over the past decades into developing group-membership systems [Bir93, CKV01]. Such systems provide processes with accurate information about which other non-failed processes are part of the system at a given time. Membership information is of great importance since it can considerably simplify the design of distributed protocols built on top of group membership.

However, implementing accurate group-membership systems is difficult and expensive, let alone their possible dependencies on the underlying system model. Therefore, many approaches make restrictive assumptions about the underlying system and, to improve performance, weak variants of group membership have been proposed. Such variants, for instance, provide processes with different views of the system at the same time and compensate for this lack of consistency with other types of guarantees, such as total ordering of messages and views.

Despite the many implementation efforts and weak variants in semantics, group-membership protocols remain expensive, especially in settings involving large numbers of participants or wide geographical dispersion. Furthermore, in large-scale systems, processes are loosely coupled by nature and it is not clear whether in such environments group membership is the right abstraction.

In this paper we advocate a new way to design large-scale systems, which does not require its members to have complete knowledge about the current system's participants. System participants are still required to know some other participants (e.g., a small set of neighbors) to avoid isolation, but they do not need to have a global view of the system. We propose a distributed system model adapted to "membership-oblivious" protocols, i.e., protocols that do not depend on the number of processes in the system, and therefore do not depend on membership. Such protocols are clearly useful when processes join and leave the system frequently, making them particularly suitable for mobile environments.

We propose to implement membership-oblivious distributed protocols using the notion of *voting currency*, and to use epidemic message propagation for interprocess communication. A *currency management system*, built on top of the underlying network (e.g., Internet or wireless mobile network), is responsible for distributing voting currency to processes, which employ it to participate in membership-oblivious protocols. The currency management system can be implemented in

several different ways depending on the underlying network. In contrast, the membership-oblivious protocols only rely on the currency abstraction and do not depend on the underlying network model.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 introduces our membership-oblivious distributed system model. Section 4 presents the currency management system and Section 5 presents concluding remarks.

2. Related Work

Reliable protocols derived from agreement problems traditionally rely on the notion of group membership. For instance, group communication systems [Bir93] order messages relative to consistent "views" agreed-upon by non-failed processes. These systems explicitly rely on group membership to implement *view-synchronous communication* and maintain strong process consistency.

Consistent group-membership protocols and related agreement protocols in general are costly and known not to scale well to large populations and wide geographical dispersion [CS93, CKV01, PS97]. Several approaches have been devised to alleviate scalability limitations.

In [Gol92], Golding proposed lightweight and scalable protocols that implement *weakly-consistency* group membership. These protocols allow temporary inconsistencies in membership views, but ensure that all processes eventually converge to a consistent view of the membership.

Epidemic protocols, also known as gossip protocols, were introduced in [DGH+87] in the context of replicated database consistency management. Since then, a wide range of epidemic-style and probabilistic protocols have been proposed for various problems, such as reliable broadcast [BHO+99, LM99, SS00, FP02, KMG03], failure detection [RMH98, GCG01], garbage collection [GHvR+97], and leader election [GvRB00]. Such protocols guarantee correctness with a very high probability and, at the same time, scale significantly better than deterministic protocols. Optimistic protocols (e.g., [GPP93, PS98, FS01]), optimized for certain runtime conditions, have also proved to be more scalable than traditional protocols with strong reliability properties. Of particular interest are epidemic-style group membership protocols [AKM01, EHG+01, GKM03], which provide each node with a partial group view used to disseminate messages in the system. Such protocols are orthogonal to our techniques. Since they do not require processes to know the complete membership of the system at any time, we could build currency management on top of them.

Quorum systems and weighted voting schemes [Gif79] are mechanisms that only require a subset of the processes to be available to ensure progress and preserve safety. As such, quorum systems are well adapted to environments where some processes frequently become unreachable. Probabilistic quorum systems [MRW97] further improve the performance and availability of quorum systems in face of failures by only ensuring consistency of replicated data with high probability. Such systems do, however, rely on consistent group membership. In [Kel99], Keleher introduced the notion of "currency" for distributed protocols as an extension of weighted voting that does not depend on membership. Keleher proposes a decentralized data replication protocol that combines epidemic propagation with voting to eventually commit updates on data items. The techniques described in this paper generalize and extend this work.

Membership-oblivious protocols are especially useful for mobile environments, where connectivity is intermittent and strong group membership is often unmaintainable. Distributed protocols targeted at mobile systems have been an active field of research recently. In [PB98], the authors propose a multi-level architecture for group communication adapted to mobile environments; group membership is implemented on top of a *proximity* layer that provides an abstract, high-level view of the underlying mobile network. Protocols for causal ordering and consensus in mobile environments, which do not explicitly rely on group membership, have also been presented in [PRS96] and [BHM99]. In this paper, we propose mechanisms for building membership-oblivious protocols that can be readily applied in, but are not limited to, mobile environments.

3. Membership-Oblivious Protocols

3.1. System Assumptions

We consider a distributed system $\Pi = \{p_1, p_2, ...\}$ composed of processes that communicate by message passing. The system is asynchronous, that is, we make no assumptions about the time it takes for processes to execute and messages to be transmitted. Processes can crash and subsequently recover, but they never behave maliciously (i.e., we do not consider Byzantine failures). Some processes may have access to local stable storage (e.g., local disk).

We define two special system events: *connect* and *disconnect*. Process p connects to the system when it

(re)starts its computation after being disconnected or when it recovers from a failure; p disconnects from the system when it voluntarily interrupts its computation or when it crashes. We assume that the system is large and composed of a variety of dynamic processes: some remain connected for long durations while others remain connected for short durations only. When a distinction is needed in the paper, we call processes in the former category *stable* and processes in the latter category *unstable*. The set of connected processes at time t is denoted by $Up(t)$. The $Up(t)$ notation is only used to simplify the presentation; no process knows all the processes connected to the system at any time.

Communication links are unreliable but fair. If sender and receiver remain connected long enough, and messages are continuously re-transmitted, they are eventually received—such communication links are also known as *fair-lossy* links [ACT00]. Our notion of communication link does not only model a physical cable connecting two computers, but also wireless connections. We denote by $neighbors(p)$ the set of processes with which process p can communicate directly. Process p can communicate directly with its neighbors and indirectly with any process q as long as there is a path between p and q.

Due to disconnections, the system may become *partitioned*. For example, if messages from processes in some subset A of the system addressed to processes in subset B have to pass by some process p and p is disconnected, then communication between processes in the different subsets is not possible. We tolerate partitions but assume that they eventually heal. In our example above, this means that either p eventually connects back, or another communication link is created between processes in A and in B.

Our system model is meant to capture a large variety of real systems since we believe that membership-oblivious protocols are useful in many different contexts. Two examples of such contexts are large Internet-based environments and mobile networks. In both cases, due to scale and frequent connections and disconnections, system membership is hard to obtain. In the Internet, stable and unstable processes can map onto servers and end-user computers (e.g., with a dial-up connection), respectively, while in mobile networks they map onto base stations and mobile hosts, respectively.

3.2. General Framework

Quorum-based protocols [Gif79] are commonly used in distributed systems to coordinate the execution of multiple processes, despite some of them being unavailable (i.e., disconnected). Provided that quorums intersect, reaching agreement among all processes of a single quorum is sufficient for preventing other processes from taking conflicting decision. Quorums promote availability because only a subset of the processes is necessary for making progress. In order to implement a quorum-based solution, however, processes have to know the current membership of the system, i.e., *which* processes are part of the system, or at least the current membership cardinality, i.e., *how many* processes are part of the system. In large and dynamic systems, obtaining membership information may be expensive if feasible at all. For such situations we need alternative ways to allow processes to know when they have gathered a quorum while lacking membership information.

In this paper we consider an alternative approach based on *currencies* instead of quorums. The basic idea is to assign to each process in the system a certain amount of currency. Membership-oblivious protocols can then be designed to rely only on the currency assigned to processes, and currency management can be performed independently of the protocol that uses it (see Figure 1).

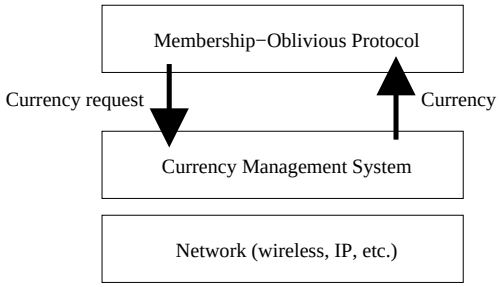

Figure 1. Membership-oblivious protocols

Currencies can be assigned to processes in many different ways, but whatever scheme is chosen, the two properties presented below are always guaranteed to hold. Hereafter, the currency assigned to process p at time t is denoted $currency_p(t)$.

P1 The currency assigned to each process at any time is a value between 0 and 1.
Formally, $\forall p \in \Pi, \forall t : 0 \leq currency_p(t) \leq 1$.

P2 The sum of all currencies assigned to connected processes at any time is at most 1.
Formally, $\forall t : \sum_{p \in Up(t)} currency_p(t) \leq 1$.

Therefore, the condition to gather a quorum is no longer dictated by the number of processes that are part of the quorum, but by the amount of currency

these processes have. In particular, an intersection between quorums is guaranteed whenever enough processes are gathered so that their aggregated currency is greater than 0.5. In principle, this is similar to weighted voting, introduced in the late 70's by Gifford [Gif79].

The contribution of this paper does not lie on the notion of currency itself—the concept, an extension of weighted voting, has been introduced elsewhere [Kel99]—but rather on the management of currencies and how it can be used to devise protocols that have been traditionally based on membership information. We also discuss issues involving the separation between membership-oblivious protocols, which use currencies, and the currency management system component.

Currency management essentially involves assigning currency to processes and reclaiming currency from processes. Even if processes knew the system membership cardinality (i.e., $|\Pi|$)—in which case each process could be trivially assigned $1/|\Pi|$ as its currency[1]—some form of currency management would still be suitable to prevent the system from blocking. Indeed, if "too many" processes are temporarily disconnected, protocols based on static currency assignment may become indefinitely blocked. Moreover, as we show later in the paper, we can improve the performance of currency-based protocol by handling currencies adequately (i.e., by applying an adequate distribution policy).

4. Currency Management

The use of process-independent currency makes it possible to develop membership-oblivious protocols. A number of non-trivial issues, however, have to be carefully dealt with for this scheme to be effective. In this section we address issues related to currency allocation, distribution policy, process failures, point-to-point currency transfer between processes, and currency redistribution.

Currency management may be dealt with in a number of ways, and they only concern the *management* of the currency itself; the actual *usage* of the currency in membership-oblivious protocols is not affected by any of the mechanisms presented below, although they have an impact on the performance of the protocol. This is a powerful characteristic of membership-oblivious protocols since performance can be improved by monitoring the execution of the system and adequately assigning currencies to processes without making any changes to the protocols that use them.

1 Notice that in this case, $\sum_{p\in\Pi} currency_p(t) = 1$, which implies $\forall t : \sum_{p\in Up(t)} currency_p(t) \leq 1$.

Each process has access to a local currency management component. We denote the currency management component at process p_i by $CurrencyMgr_i$. It handles issues related to currency management such as *point-to-point currency transfer* and *currency redistribution*. The former denotes the transfer of currency between two processes, while the latter resets the currency distribution in the entire system. The currency manager provides two functions: (1) getting the actual amount of currency and (2) releasing all the currency. Function (1) returns the actual amount of currency that is allocated to the particular process. Function (2) returns all the currency to the process that has originally issued it. This abstraction allows us to implement different currency management policies transparently to the process holding the currency. However, closer interaction may take place between the currency manager and the application, as the application semantics may be instrumental to devise an adequate currency distribution policy.

4.1. Currency Allocation

We assign to stable processes the role of collecting and distributing the currency in the system. They can be, for instance, base stations in a wireless network or highly-available servers in a wired network. We do not make any assumptions about unstable processes' availability and location. They can disconnect and reconnect infinitely often, they can disappear from the system altogether, and in mobile environments they can even change their physical location inside the system.

Stable processes can be seen as a core group of processes that are closely coupled: they maintain (not necessarily complete) information about other stable processes in the system and can communicate reliably with each other.[2] We also assume that the currency manager of a stable process has access to stable memory, so that it can recover its state after a failure. Note that stable processes are only needed for currency management. Membership-oblivious protocols that *use* the currency manager do not necessarily need any assumptions about process stability.

To simplify the explanation, we assume some initial configuration in which each stable process is assigned a certain amount of the global currency and no currency is assigned to unstable processes. We also consider that

2 Reliable communication is built on top of unreliable links by re-transmitting lost messages. Unstable processes can also in principle implement reliable communication through similar means, but since they are not expected to remain connected for long periods, re-transmission is more limited than with stable processes, and thus we do not assume that unstable processes can communicate reliably.

the sum of all the currency assigned to stable processes is 1 (Property P2 in Section 3).

Stable processes can use their currency to participate in the execution of a membership-oblivious protocol, or they can distribute part of their currency to other processes. Typically, a stable process would distribute some of its currency to its neighbors. In a wireless network, a base station would distribute currency to mobile processes within its cell (i.e., the area within which it can communicate with mobile processes through wireless communication links). In a wired network, a highly-available server could distribute its currency to geographically-close processes.

In general, the more currency is distributed, i.e., the smaller the amount of currency per process, the more fault-tolerant the application becomes. On the other hand, the performance decreases, as more messages are needed before a process can gather a certain amount of currency.

A stable process should not distribute more currency that it currently owns. Although obvious, this observation makes currency allocation a challenging problem since stable processes must avoid running out of currency. A simple solution to the problem is to distribute currency sparsely, for example by always allocating a fraction (e.g., 1/10) of the currency currently available. Therefore, it does not matter how many processes request currency; stable processes will always be able to provide some.

The problem with this allocation strategy is that it is not fair: late processes will receive less currency than early processes. In fact, if some stable process always gives $1/c$ of its initial currency K, the n-th process will only receive $K(c-1)^{n-1}/c^n$. A fair scheme would require the number of processes requesting currency to be known beforehand by the stable process.

One approach to increase the fairness of the aforementioned strategy without knowledge of the process population is to ask processes to permanently "renegotiate" their currency. From time to time and unless it fails, each unstable process gives back its currency to the stable process that originally gave it away. The stable process adds up the received currency to its current currency and immediately re-assigns a fraction of the result to the unstable process. Provided that the number of unstable processes that contact a stable process does not change for a certain duration, and they all keep communicating with the stable process, then the unstable processes will eventually have a similar amount of currency. For example, if a single stable process with currency 1 distributes 10% of its current currency to 10 processes, after 2 currency re-negotiations the difference between the currency owned by any two processes is less than 3%!

There are yet other approaches to deal with currency allocation. As a last example, we can have stable processes periodically invalidate all previously allocated currency, and then redistribute the new one. We elaborate on this approach in Section 4.3 as it also helps deal with currency that disappears from the system when processes fail.

4.2. Currency Distribution Policy

The currency distribution policy should take into consideration the network topology, location of processes, available resources such as bandwidth, CPU or memory, and etc. Ideally, it would rely solely on the currency management system. However, more adequate currency distribution policies may be also based on application semantics. Assume, for instance, an application that relies on atomic broadcast (or total order broadcast). Depending on the application semantics, certain processes may broadcast significantly more messages than others. Consequently, it may be more efficient to distribute the currency in such a way that it is located close to these processes.

Geographical location plays an important role in a wireless network. When a process (i.e., mobile process) moves from one location (base station) to another, it should keep its currency; when it disconnects, however, it should give back its currency to some stable process. By doing so, as processes move from one location to another, the currency distribution will reflect the processes distribution, assigning more currency to densely populated regions, regardless of the strategy used to initially assign currencies—better initial allocation strategies may however speed up the process. The motivation for concentrating currency in densely populated locations is to reduce the average latency of membership-oblivious protocols: locations with larger population tend to send more messages and execute more distributed protocols (assuming that processes have similar characteristics); by concentrating currencies in such places, processes can quickly reach the currency threshold necessary to terminate their protocols.

4.3. Process Failures

The failure of unstable processes may hamper the execution of the system (note that voluntary disconnections are not much of a problem, as discussed above). In this section, we show that failures may lead to blocking and we discuss approaches to prevent blocking.

4.3.1. The Blocking Problem Process failures may lead to some currency becoming unavailable and

consequently hamper progress of currency-based protocols. We distinguish between the failure of (1) stable and (2) unstable processes. Case (1) is generally less critical since stable processes usually have access to stable memory and recover the state of the currency manager after recovering. The currency of a failed stable process is eventually returned to the system. Hence, the failure of a stable process may slow down or prevent progress in the execution (i.e., blocking), but only temporarily during the process' downtime period. Clearly, even temporary blocking may not be appropriate for certain applications, especially if the processes have long downtimes.

When an unstable process (with no stable storage) fails, it may not be able to remember how much currency it had prior to a failure if it ever recovers. In general, currency may be lost each time a process without stable storage fails. In the extreme case, failures may cause a membership-oblivious protocol to block if there is not enough currency in the system to reach the threshold necessary for protocol termination. In contrast to case (1) above, blocking here is permanent, unless a mechanism is devised to recover lost currency.

4.3.2. Preventing the Loss of Currency To prevent currency loss, the currency of the failed process must be reclaimed and assigned to some non-failed process. Unfortunately, in an asynchronous system, it is impossible to distinguish between a slow and a failed process [FLP85]. As a consequence, a process may revoke the currency of another process it considers failed, although the latter is still operational. Solving such a problem requires additional assumptions about the system (i.e., timing assumptions). In the following, we describe two schemes for currency recovery that can be implemented in our model: (1) logging the actual currency of unstable processes on a stable process, and (2) relying on a leasing approach.

Currency Logging. In this approach, the amount of currency is logged on a stable process (generally the stable process that has issued the currency to the unstable process). Upon recovery, the unstable process has to find the stable process that contains its log entry. Unfortunately, no guarantees are given about whether the unstable process eventually finds the stable process. Moreover, the stable process may be down and the unstable process deprived of its currency until the stable process recovers. Both problems can be dealt with by having stable processes replicate the information about unstable processes in several stable processes. Finally, this approach can only be effective if the currency assigned to the unstable process does not change too often.

Leased Currency. Another approach is to limit the validity of the currency given to unstable processes (i.e., similarly to a *lease*). Currency would be tagged as being usable only for a certain duration. Using time, however, requires assumptions about clock synchronization. Alternatively, the validity of currency can be combined with the membership-oblivious application running on top of the currency management. For example, if the application is based on "asynchronous rounds," (e.g., [FP02]) then the validity of a currency can be expressed in terms of number of rounds. Once currency has expired, it is automatically recovered by the originating stable process and re-distributed to other processes upon request.

With this scheme, an unstable process p_j would receive some currency c_j with expiration round of r from process p_i, where r is some round after p_i's current round. Before giving the currency to p_j, p_i stores the difference $c_i - c_j$ in stable storage, as well as the information that c_j expires at round r. In the worst case, if p_j fails, some amount of currency may be unavailable until round r, but afterwards, the currency c_j is automatically reclaimed. If p_i fails, however, then c_j may be lost until p_i recovers. The risk of temporary blocking can be decreased by allocating the initial currency to a large number of stable processes.

4.4. Currency Transfers

Point-to-point currency transfer can occur between two stable processes, or between a stable and an unstable process. The former exchange occurs when a new stable process is added to the system. The new process starts with no currency, but can ask other stable processes for some of their currency. Point-to-point currency transfer between stable and unstable processes is the classical transfer mechanism previously discussed, with stable processes serving as currency distributors for the unstable processes.

Assume that two processes p_i and p_j start with currencies c_i and c_j, respectively. After a currency transfer, they hold currencies c'_i and c'_j. Ideally, the sum of the currencies of p_i and p_j should be preserved, i.e., $c_i + c_j = c'_i + c'_j$. Clearly, if this is the case, Property P2 is preserved. Unfortunately, in an asynchronous system where processes can fail, this is not possible to ensure. Indeed, currency transfer must be performed atomically to conserve the amount of currency under various failure patterns. As stable processes are assumed to have stable storage, we can use traditional distributed atomic commitment protocols[3] for the currency trans-

3 Note that such protocols also need timing assumptions to guarantee liveness.

fer.

If the systems behaves in truly asynchronous manner, with no timing assumptions, we can still implement currency transfers with the relaxed guarantee that $c'_i + c'_j \leq c_i + c_j$. Again, this may lead to the loss of currency and risks of blocking in extreme cases. As discussed in Section 4.3, the probability of blocking can be further reduced by using leased currency.

4.5. Currency Redistribution

From time to time, it may be necessary to redistribute the currency in the system, i.e., to reset the system to some initial state. The main motivations for doing so are to reclaim lost currency that cannot be recovered by other means, and to distribute currency evenly when the currency ownership in the system has degenerated over time into an unfavorable distribution. Redistributing currency can be performed if a set of processes with a certain currency threshold agree on a new distribution, and generate new currency. Clearly, this requires some sort of membership information. Indeed, at least a set of stable processes that could distribute currency needs to be known. If messages are received containing old currency, the senders are notified that their currency has expired and they need to obtain new currency.

5. Final Remarks

Message ordering abstractions, and more specifically group communication protocols, are very useful for the design of reliable distributed systems. Such protocols traditionally rely on the notion of group membership, which is known not to scale well to large populations and wide geographical dispersion. Group membership is also prohibitively expensive in mobile environment, where processes suffer from frequent disconnections and re-connections. Protocols that do not depend on membership are better candidates for such environments.

In this paper, we have proposed a distributed system model adapted to "membership-oblivious" protocols, i.e., protocols that do not depend on membership. A *currency management system*, implemented on top of the underlying network, is responsible for distributing voting currency to processes, which employ it to participate in membership-oblivious protocols.

There are striking commonalities between currency management discussed in this paper and the functioning of the global economy. For instance, stable processes that distribute currency act like the countries' central banks. In the future, we plan to exploit models developed in the context of the global economy for currency management. These models could enable forecasts as to how the currency distribution develops with time. Moreover, we are considering extending our approach with further concepts from the global economy, such as local currencies and inflation.

References

[ACT00] M.K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[AKM01] A.J.Ganesh, A.-M Kermarrec, and L. Massoulie. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *3rd International workshop on Networked Group Communication*, London, UK, November 2001.

[BHM99] N. Badache, M. Hurfin, and R. Macedo. Solving the consensus problem in a mobile environment. In *Proceedings of the 1999 IEEE International Performance, Computing, and Communications Conference (IPCCC'99)*, pages 29–35, Phoenix, USA, February 1999.

[BHO+99] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[Bir93] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[CKV01] G.V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[CS93] D.R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 44–57. ACM Press, 1993.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, BC, Canada, August 1987.

[EHG+01] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, Newport, Rhode Island, USA, August 2001.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one

faulty process. *Journal of the ACM*, 32(2):217–246, 1985.

[FP02] P. Felber and F. Pedone. Probabilistic atomic broadcast. In *Proceedings of the 21st Symposium on Reliable Distributed Systems*, Osaka, Japan, October 2002.

[FS01] P. Felber and A. Schiper. Optimistic active replication. In *21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 333–341, Phoenix, AZ, April 2001.

[GCG01] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, Newport, Rhode Island, USA, August 2001.

[GHvR+97] K. Guo, M. Hayden, R. van Renesse, W. Vogels, and K. P. Birman. GSGC: An efficient gossip-style garbage collection scheme for scalable reliable multicast. Technical Report TR97-1656, Cornell University, Computer Science, December 1997.

[Gif79] D.K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.

[GKM03] A.J. Ganesh, A.-M Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, February 2003.

[Gol92] R.A. Golding. *Weak-consistency group communication and membership.* PhD thesis, University of California, Santa Cruz, December 1992. Number UCSC-CRL-92-52.

[GPP93] R.G. Guy, G.J. Popek, and T.W. Page Jr. Consistency algorithms for optimistic replication. In *1st IEEE Int. Conference on Network Protocols*, October 1993.

[GvRB00] I. Gupta, R. van Renesse, and K. P. Birman. A probabilistically correct leader election protocol for large groups. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 89–103, Toledo, Spain, October 2000.

[Kel99] P.J. Keleher. Decentalized replicated-object protocols. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 143–151, Atlanta, USA, May 1999.

[KMG03] A.-M Kermarrec, L. Massoulie, and A.J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, March 2003.

[LM99] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. Technical Report CS1999-0622, University of California, San Diego, Computer Science and Engineering, June 1999.

[MRW97] D. Malkhi, M. Reiter, and R. Wright. Probabilistic quorum systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–273, August 1997.

[PB98] R. Prakash and R. Baldoni. Architecture for group communication in mobile systems. In *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 235–242, West Lafayette, USA, October 1998.

[PRS96] R. Prakash, M. Raynal, and M. Singhal. An efficient causal ordering algorithm for mobile computing environments. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 744–751, Hong Kong, May 1996.

[PS97] R. Piantoni and C. Stancescu. Implementing the swiss exchange trading system. In *Proc. of 27th Int. Symposium on Fault-Tolerant Computing (FTCS)*, pages 309–313, Piscataway, NJ, June 1997.

[PS98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In *12th. Intl. Symposium on Distributed Computing (DISC'98)*, pages 318–332, Andros, Greece, September 1998. Springer Verlag, LNCS 1499.

[RMH98] R. Van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, Cornell University, Computer Science, May 1998.

[SS00] Q. Sun and D. Sturman. A gossip-based reliable multicast for large-scale high-throughput applications. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, New York (USA), June 2000.