The "Art" of Programming Gossip-based Systems*

Patrick Eugster
Purdue University
West Lafayette, IN, USA
peugster@cs.purdue.edu

Pascal Felber
University of Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

Fabrice Le Fessant
INRIA
Orsay, France
Fabrice.Le_fessant@inria.fr

ABSTRACT

How does one best go about building actual gossip-based protocols? Trying to answer this question has brought us to address two preliminary questions, namely (1) what the intrinsics of such systems or protocols are, and (2) what kind of applications would in the end be built on top of such protocols. We address the first question by arguing that gossip-based protocols are all built following one and the same pattern, and describing three building blocks which we claim are used to support this recurrent pattern—most notably a source of randomness. We validate these claims by devising simplified versions of well-known protocols, in a layered fashion, on top of a conceptual interface describing these basic services. The second question is addressed by arguing that gossip-based protocols exhibit some probabilistic or imperfect flavor (e.g., probabilistic or partial completion), and by proposing to take such probabilistic behavior into account when devising interfaces for applications building on top of gossip-based protocols. We argue for inherent support for these probabilities in the programming model.

1. INTRODUCTION

Although gossip-based (or epidemics-style) approaches have been already employed in many early distributed algorithms, it is recently through the advent of large-scale and peer-to-peer systems that gossip-based programming has become popular.

An emblematic example of the use of gossip-based algorithms is the robust and scalable propagation of information in distributed systems [3]. A process that wishes to disseminate a new piece of information to the system does not send it to a server, or a cluster of servers, in charge of forwarding it, but rather to a set of other peer processes, chosen at random. In turn, each of these processes does the same, and also forwards the information to randomly selected processes, and so forth.

The principle underlying this information dissemination technique mimics the spread of a rumour among humans via gossiping or the spread of infectious diseases as *epidemics*; hence the names *gossip*-based and *epidemic* dissemination algorithms.

Once started, epidemics are hard to eradicate: a few people infected by a contagious disease are able to spread it, directly or indirectly, to a large population. Epidemics are resilient to *failures* in the infection process. That is, even if many infected people die before being able to transmit the disease, or are immunised, the epidemic is still *reliably* propagated over populations.

Gossip-based dissemination algorithms are simple and easy to deploy. In addition to their attractive scalability promises, they exhibit a very stable behaviour even in the presence of a high rate of link and/or process failures. There is no single point of failure and the reliability degrades gracefully with the number of failures. A large amount of research has been devoted to observing, analysing, and devising mathematical theories for epidemics. This are at the disposition of designers of gossip-based algorithms

Applying the epidemic idea to disseminate information among a large number of processes with a dynamic connection topology is thus very appealing [15]. Not surprisingly, the use of gossip-based algorithms has been explored in various other potentially large-scale applications such as database replication [5], failure detection [22], data aggregation [11], resource discovery and monitoring [21], or publish/subscribe [4].

But how does one best go about *building* actual gossip-based protocols? Trying to answer this question has brought us to address two preliminary questions, namely (1) what the intrinsics of such systems or protocols are, and (2) what kind of applications would in the end be built on top of such protocols. We attempt to provide answers to these questions in the following, before proposing corresponding support for gossip-based programming and outlining first steps that could be used for corresponding research agendas.

*While the authors claim that gossip-based programming may be equated to an "art", they by no means pretend mastery of that art.

2. DEVELOPING GOSSIP-BASED PROTOCOLS

In order to propose better support for gossip-based protocols, we first present a characterization of these protocols, based (1) on a recurrent pattern underlying these protocols, and (2) on basic services they require.

A recurrent pattern of gossip-based interaction

Based on our observation of existing gossip-based protocols found in the literature and deployed, we conjecture that all gossip-based protocols follow the same general interaction pattern. Locally, an application performs the following steps:

> Upon the reception of an event (which can be a message or the expiration of a timer), the process can:
>
> 1. query and/or modify its local neighborhood,
> 2. modify the event and/or generate a new event, and
> 3. select the neighbor(s) to send the event to.

One can argue that this pattern is generic enough to describe the behavior of *any* distributed protocol, gossip-based or not, or that, inversely, any distributed protocol can be refactored to abide to this pattern. We believe that the intrinsics of gossip-based protocols are manifest in the above pattern in that, at least at this point, the local neighborhood is neither assumed nor necessarily intended to be "complete", not even eventually. In addition, many gossip-based protocols are characterized by their non-determinism (notably in neighbors selection when communication) and, often, by their periodic operation.

Core services for gossip-based protocols

The crucial distinction to other distributed protocols crystalizes in combination with the following three building blocks, which we claim to be underlying any gossip-based protocol:

(a) a source of randomness,

(b) a set of neighboring processes, and

(c) communication channels to these neighbors.

So far, the model is intentionally generic, in order to accommodate different protocols and underlying services and infrastructures with varying semantics. For instance, the precise semantics of communication channels are not detailed at this point, as they may vary according to the underlying network etc. The most salient of these three building blocks is the source of randomness. While not *all* protocols that are referred to in the literature as gossip-based ones do indeed make explicit use of randomization, there are definitely many typical examples of gossip-based protocols which do involve explicit randomization. As a matter of fact, randomness is key to the robustness and scalability of many gossip-based protocols (e.g., [3, 10]). Non-determinism may also "emerge" as a consequence of the impossibility of tracking all actions and reactions of a large set of nodes. In contrast to many deterministic protocols, gossip-based ones do not attempt to bring perfect order and structure into the system, not even evetually.

An API for gossip-based protocols

We have validated this model by defining a minimal Application Programming Interface (API) for these building blocks and implementing higher level services in a layered manner. We have also implemented several gossip-based protocols, notably: automatic neighbour management, gossip-based broadcast following the Renyi-Erdös graph model [6, 16], random walk [17], Gnutella search protocol [9], and CYCLON [23].

All our examples are presented as informal pseudo-code with various simplifications to clarify the presentation. Actual implementations are likely to be more involved. Yet, the general principle will remain unchanged.

Interface 1 Randomness

```
1: fun RANDOM(range) : int        ▷ Returns a random number
```

The first part of the API deals with randomness. It allows algorithms to access a random number generator and "flip a coin", as often required by algorithms with probabilistic properties like the Ben-Or consensus algorithm [2].

Interface 2 Neighborhood

```
1: fun GETNEIGHBORS(nb) : peer list   ▷ Returns list of neighbors
2: fun ADDNEIGHBOR(peer)              ▷ Add new neighbor
3: fun REMOVENEIGHBOR(peer)           ▷ Remove neighbor
```

The second part of the API deals with peer neighborhood. The neighborhood of a peer p is a list of other peers with whom p is interested to work. It can be seen as a weak version of a group-membership system, where the local view only contains a subset of the system's processes. To simplify presentation, we assume that we can ask for a *random* subset of a peer's neighbors by calling GETNEIGHBORS with an integer parameter specifying the size of the expected subset.

Interface 3 Communication

```
1: fun SEND(msg) TO dest              ▷ Send message
2: fun ADDMSGHANDLER(msg, handler)    ▷ Call back upon message
3: fun ADDTIMER(delay, handler)       ▷ Call back at given time
```

The final part of the API deals of course with communications. A peer can send (non-atomically) a message to a set of peers, receive a message from another peer, and call a function after a given delay. All these operations are obviously non-deterministic, as no guarantee is given on the reliability of communication channels nor timers.

Illustration

We now convey our claims by illustrating how our minial API can be used to construct well-known gossip-based protocols. Note that the examples shown below have been chosen to be representative of different types of gossip-based protocols that use various services provided by the underlying API, but they do not exhaustively cover the full spectrum of gossip-based protocols.

Figure 1 provides an overview of the illustrated building blocks, and how they *can* (the top-most components *need not* be built on top of the third layer) be layered on top of each other.

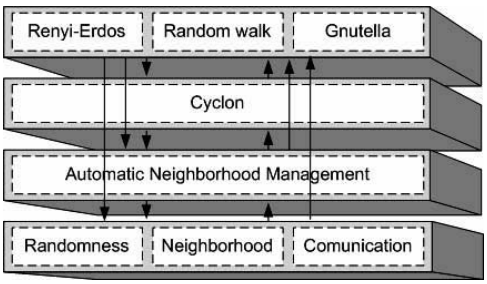

Figure 1: Illustration of core services and APIs

The first protocol that we consider implements automatic neighbor management by replacing failed neighbors (see Algorithm 4). We use timers to periodically send heartbeat messages to our neighbors. When we detect that a neighbor failed to send a heartbeat, we replace that neighbor by another one discovered using a random walk (to be described shortly). This protocol would typically be used in unstructured overlays to maintain a constant number of neighbors per node.

Algorithm 4 Automatic Neighbor Management

```
1: addTimer(Δ, Heartbeat)
2: addMsgHandler(PING, PingHandler)

3: function HeartBeat()
4:     for all n ∈ GetNeighbors() do
5:         if last[n] + TIMEOUT < Now() then      ▷ Ping not received
6:             Failure(n)                          ▷ Remove neighbor
7:         else
8:             send(PING) to n                     ▷ Send new ping
9:         end if
10:    end for
11:    addTimer(Δ, Heartbeat)
12: end function

13: function PingHandler(n, msg)                   ▷ Receive ping from n
14:    if n ∈ GetNeighbors() then
15:        last[n] ← Now()
16:    end if
17: end function

18: function Failure(n)                            ▷ Did not receive ping from n
19:    removeNeighbor(n)
20:    repeat                                      ▷ Find replacement neighbor
21:        n ← RandomNode()                        ▷ Use random walk
22:    until n ∉ GetNeighbors()
23:    addNeighbor(n)
24: end function
```

Our second example is a gossip-based broadcast that follows the Renyi-Erdös graph model [6, 16], in which each peer that receives a message for the first time forwards it to a number of neighbors logarithmic in the network size (see Algorithm 5). Note that there exist several approaches for estimating the size of a dynamic network (e.g., [13, 18, 20]).

Algorithm 6 illustrates a random walk with a bounded distance. A counter in the message is decreased at each hop before the message is forwarded to a random neighbor. When the counter becomes null, i.e., the walk ends, the originator receives a notification from the final peer and replaces a random neighbor by the peer discovered by the random walk. Such a protocoal can be used to continuously reshuffle the network.

Algorithm 5 Renyi-Erdös

```
1: addMsgHandler(RENYI-ERDOS, Renyi-Erdos)
2: Received ← ∅
3: N ← ApproximateSystemSize()

4: function Renyi-Erdos(n, msg)                    ▷ Receive msg from n
5:     if msg ∉ Received then
6:         send(RENYI-ERDOS, msg) to GetNeighbors(log(N))
7:         Received ← Received ∪ {msg}
8:     end if
9: end function
```

A simplified version of the Gnutella protocol [9] is shown in Algorithm 7. A search is initiated by flooding a region of the network (using a degree and a diameter of 7). A peer that receives the query for the first time searches its local database and returns local matches, if any, to the neighbor that sent the query. Hence, replies follow the reverse path of queries. The originator of the search collects and prints replies. This example illustrates how exceedingly gossip-based protocols can provide powerful and sophisticated funtionality when executed by a large number of nodes.

Algorithm 6 Random walk

```
1: addMsgHandler(WALK, Walk)
2: addMsgHandler(WALK-DONE, WalkDone)

3: function WalkStart(distance)                    ▷ Start random walk
4:     msg[hops] ← distance
5:     msg[origin] ← p
6:     msg[path] ← []
7:     send(WALK, msg) to GetNeighbors(1)
8: end function

9: function Walk(n, msg)                           ▷ Receive msg from n
10:    hops ← msg[hops]
11:    path ← msg[path]
12:    hops ← hops − 1
13:    path ← [path,self()]
14:    msg[hops] ← hops
15:    msg[path] ← path
16:    if hops = 0 then
17:        send(WALK-DONE, msg) to msg[origin]
18:    else
19:        send(WALK, msg) to GetNeighbors(1)
20:    end if
21: end function

22: function WalkDone(n, msg)                      ▷ Return from random walk
23:    for all p ∈ msg[path] do
24:        echo p
25:    end for
26:    if n ∉ GetNeighbors() then                  ▷ Replace random neighbor
27:        removeNeighbor(GetNeighbors(1))
28:        addNeighbor(n)
29:    end if
30: end function
```

Finally, Algorithm 8 illustrates the CYCLON protocol [23] that periodically reshuffles the neighborhood of the peers. Each peer p repeatedly initiates a neighbor exchange operation by (1) selecting a random subset of neighbors and a random peer q from that subset; (2) replacing q by p in the subset; (3) sending the subset to q; (4) receiving a subset of q's neighbors from q; and (5) updating p's neighbors using the subset received from q. Again, we observe that complex functionality (when considerred from a global perspective) can be implemented by remarkably simple gossip-based protocols.

Although the set of gossip-based protocols used for illustration is far from exhaustive, it covers a wide range of epidemic behaviors and interaction patterns. This tends to indicate that the core services identified could easily support other gossip-based protocols. One can easily construct higher-level building blocks useful for a wide range of applications, such as a peer sampling service [14]. It is important to consider that all the algorithms discussed here use a source of randomness, which we believe is a distinguishing feature of gossip-based algorithms.

Algorithm 7 Gnutella

```
1: addMsgHandler(QUERY, Query)
2: addMsgHandler(QUERY-DONE, QueryDone)
3: Received ← ∅

4: function QueryStart(query)                    ▷ Send query through network
5:     msg[ttl] ← 7
6:     msg[origin] ← p
7:     msg[query] ← query
8:     msg[id] ← newGlobalID(p)
9:     send(QUERY, msg) to GetNeighbors(7)
10: end function

11: function Query(n, msg)                       ▷ Receive query from n
12:     id ← msg[id]
13:     if {id, −} ∉ Received then
14:         Received ← Received ∪ {id, n}
15:         ttl ← msg[ttl]
16:         if ttl > 0 then
17:             msg[ttl] ← ttl − 1
18:             send(QUERY, msg) to GetNeighbors(7)
19:         end if
20:         query ← msg[query]
21:         hits ← SELECT query FROM files
22:         if hits ≠ ∅ then
23:             msg[result] ← hits
24:             msg[server] ← p
25:             send(QUERY-HIT, msg) to n
26:         end if
27:     end if
28: end function

29: function QueryDone(n, msg)                   ▷ Return query hit
30:     id ← msg[id]
31:     if {id, n_i} ∈ Received then
32:         send(QUERY-HIT, msg) to n_i
33:     end if
34:     if msg[id] = p then
35:         echo msg[result] @ msg[server]
36:     end if
37: end function
```

3. DEVELOPING GOSSIP-BASED APPLICATIONS

No matter the precise reason for which gossip-based solutions are employed—scalability, efficiency, reliability—their non-deterministic nature ends up reflecting through probabilistic guarantees and behavior. We advocate for making this probabilistic nature explicit, rather than employing gossip-based building blocks as if they provided deterministic guarantees.

Employing probabilistic protocols

We view deterministic protocols as special cases of probabilistic protocols. They just succeed with probability $P = 1$ or fail. We envision a system where probabilities are reflected in the interface, i.e., primitives for distributed interactions are parametrized with some probabilistic measures. We distinguish two places in which these measures can appear, namely as (a) input values, and/or (b) output values.

Algorithm 8 CYCLON

```
1: addTimer(ShuffleStart(), 60s)
2: addMsgHandler(SHUFFLE, Shuffle)
3: ℓ ← shuffle length

4: function ShuffleStart(query)                  ▷ Start shuffling neighbors
5:     peers ← GetNeighbors(ℓ)
6:     q ← Random(peers)
7:     peers ← peers \ {q} ∪ {p}
8:     msg[newpeers] ← peers
9:     msg[origin] ← p
10:    send(SHUFFLE, msg) to q
11: end function

12: function Shuffle(n, msg)                     ▷ Receive shuffling request from n
13:     new ← msg[newpeers]
14:     origin ← msg[origin]
15:     if origin = p then                                  ▷ This is the reply
16:         old ← msg[oldpeers]
17:     else                                                ▷ This is the request
18:         old ← GetNeighbors(ℓ)
19:         msg[newpeers] ← new
20:         msg[oldpeers] ← old
21:         send(SHUFFLE, msg) to n
22:     end if
23:     new ← new \ {p}
24:     for all n_i ∈ new do
25:         addNeighbor(n_i)
26:     end for
27:     for all n_i ∈ old do
28:         if |GetNeighbors()| > MAX-DEGREE then
29:             removeNeighbor(n_i)
30:         end if
31:     end for
32: end function
```

More precisely, any primitive or operation triggering a gossip-based protocol can benefit from input values, which can be used to characterize the expected or required quality of the protocol result. Such values can help the protocol adjust certain of its own parameters in an attempt to satisfy the demand. Output values are the dual of input values; they reflect the imperfect nature of gossip-based protocols by conveying feedback on the outcome of a gossip-based primitive, e.g., an estimation of the outcome, or a lower bound on the partial completion.

Take the example of a gossip based broadcast, where an input probability specifies a lower bound on the desired coverage *ratio* (say $50+\%$) or, alternatively, the probability that *all* processes receive the message (cf. [8]). An example of output "probability" in this case could consist in returning information at a given point about the fraction of processes known to have been reached by then in the former case, or an estimation of the probability that all processes indeed received the message in the latter case. As an alternative, one could even imagine being continuously updated about the quality of such "partial" results, and automatically triggering some action as soon as a desired or required threshold is reached.

We conjecture that making probabilities explicit in the application could also help developers embrace gossip-based protocols. Classical probabilistic protocols behave well most of the times, but there is always a chance that things turn bad; we believe that such considerations are a hindrance to a wider adoption of gossipping. Many applications can deal with "imperfect" behavior given that the degree of imperfection is known or a lower bound can be guaranteed. For instance, a distributed system may need to disseminate infor-

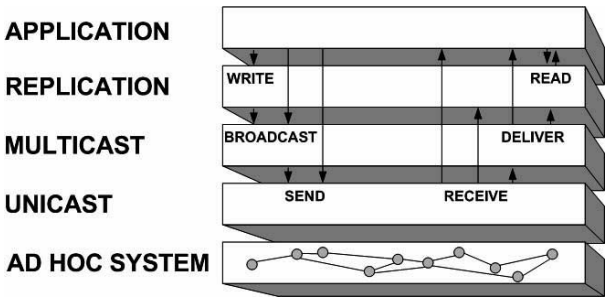

Figure 2: Combining probabilistic components

mation such that a majority of the peers must be reached to preserve consistency, but liveness is increased if more peers receive the information.

Reasoning about probabilities

Several authors have described protocols with probabilistic guarantees, and composed these. For instance, Eugster has proposed in [7] a system built by successive layering of probabilistic components akin to the ones in Figure 1—communication channels with probabilistic transmission guarantees, probabilistic broadcast, and probabilistic quorum replication (see Figure 2). An end application of this stack is a distributed certification authority. Probabilistic specifications for each layer have been provided, illustrating how a probabilistic property in a core component leads to probabilistic properties of any component build on top.

Composing probabilistic components

Can one develop inherent programming language support to promote probabilistic input as well as output values? In the context of program proving, several authors have proposed foundations for reasoning also about probabilistic components. McIver and Morgan [19] for instance introduce probabilistic guarded commands and probabilistic loops as fundamental building blocks. Probabilities defy the traditional way of reasoning about programs and proving their correctness. In axiomatic semantics for instance, it is common to work with Hoare-triples[1] of the form $\{A\}f\{B\}$, where a function or operation f may execute if its precondition A holds, guaranteeing its postcondition B subsequently. If one can prove that B implies C, where C is the precondition for some second operation g, reasoning can be performed stepwise in a scenario as the following:

$$\{A\}f\{B\}g\{D\}...$$

Suppose now f is part of a probabilistic protocol. In this case, B will be satisfied with a probability only, or, if f provides feedback on its success rate, B can be integrating that rate, and a simple conditional branch can be used to guard g.

Introducing inherent support for such branching for instance only seems to make sense if all probabilistic components do manifest input and output probabilities with the same semantics. Given the wide variety of different (probabilistic) properties of existing protocols, we don't think this path can be straightforwardly pursued. In particular, only few protocols provide output values, i.e., feedback on their completion.

[1]Axiomatic semantics are sometimes also referred to as Hoare-logic [12].

Dealing with multiple outputs

The possibility of a probabilistic component yielding different outcomes in response to its invocation leads to the issue of dealing with these multiple outputs. This situation is reminiscent of exception handling mechanisms in programming languages, in particular object-oriented ones. In earlier languages without support for such exceptions, as in C, different outcomes of a function, including outcomes reflecting errorenous states or failures, would simply be achieved by returning specific values representing such states, for instance simple integer values (e.g., `-1` for an aborted operation). Inherent (static) support for exceptions in programming languages such as Java have improved both readability and safety of code with respect to such "design pattern"-like approaches [1].

Programmers could strongly benefit from similar support for programming with probabilistic components, when providing information representing probabilistic output. Rather than signallng a potential set of exceptions that a function can return, we foresee a set of probabilities that are provided as part of the outcome of such a primitive. Such probabilities can be returned alongside with a "classic" return value, or only in absence thereof; several of them can be returned as outcome of the invocation of the probabilistic component. We advocate for `try... catch`-like constructs to support safe branching based on different outcome scenarios. The conditions for such branching are then application-defined criteria based on the potentially multiple output values (probabilities), for example representing thresholds triggering different subsequent behavior.

Encapsulating probabilistic behavior

In functional programming languages such as Haskell, *monads* [24] have been proposed as programming language constructs to separate the handling of exceptional outcomes from the "common" execution (evaluation) path, and to avoid "partial" computations if at some point along such a path an exceptional state arises. Such computations are considered to be side-effects in this context. Monads have been used to explicitly deal with such side-effects by deferring any computation until a "safe point". I/O have been in this sense modeled as *desired* side-effects, which are however also to become effective only at precise points in time.

In the same sense, a monadic type system could allow for a cleaner separation of pure logical functionality of a program from its probabilistic, imperfect, nature. How such support would exactly look, and how probabilistic behavior could be expressed in it, requires however further investigation.

4. CONCLUSION

Gossip-based protocols are now important building stones in large-scale decentralized systems, such as peer-to-peer networks. In this paper, we presented two important aspects in programming such gossip-based protocols.

First, we proposed a very simple low-level interface, suffisant enough to write gossip-based protocols. Although this interface might appear standard in distributed systems, one of its key features is the inheritent randomness and nondeterminism of its primitives: indeed, more than other protocols, gossip-based protocols must deal with a very small knowledge of the system and very weak assumptions on the reliability of its components (communication channels and hosts), requiring to use a large amount of probabilistic operations. Through multiple examples, we showed that this interface can express well-known gossip-based protocols, from the simplest ones to more complicated ones.

Finally, we claimed at a higher-level that functionalities implemented over gossip-based protocols should be able to provide either the user or the system with probabilities of success. This favors the design of a language with "probabilistic primitives", where programs with underlying probabilities would be easy to develop and reason about. Such programs would be good candidates to manage and use gossip-based protocols and functionalities.

5. REFERENCES

[1] D. Ancona, G. Lagorio, and E. Zucca. A Core Calculus for Java Exceptions. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 16–30, October 2001.

[2] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement prototols. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, August 1983.

[3] 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.

[4] P. Costa, M. Migliavacca, G.P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 552–561, March 2004.

[5] 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 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12, August 1987.

[6] P. Erdös and A. Rényi. On the evolution of random graphs. *Mat. Kuttató. Int. Közl.*, 5:17–60, 1960.

[7] P. Eugster. *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*, chapter Reliable Computing in Ad-hoc Networks, pages 219–230. CRC Press, August 2005.

[8] P. Eugster, R. Guerraoui, and P. Kouznetsov. $\Delta-$Reliable Broadcast: A Probabilistic Measure of Broadcast Reliability. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 636–643, March 2004.

[9] Gnutella. http://www.the-gdf.org.

[10] I. Gupta, A.-M. Kermarrec, and A.J. Ganesh. Efficient Epidemic-Style Protocols for Reliable and Scalable Multicast. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 180–189, October 2002.

[11] I. Gupta, R. van Renesse, and K.P. Birman. Scalable Fault-Tolerant Aggregation in Large Process Groups. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 433–442, June 2001.

[12] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), October 1969.

[13] K. Horowitz and D. Malkhi. Estimating network size from local information. *Inf. Process. Lett.*, 88(5):237–243, 2003.

[14] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of Middleware*, pages 79–98, October 2004.

[15] R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking. Randomized Rumor Spreading. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 565–574, November 2000.

[16] A.-M. Kermarrec, L. Massoulie, and Ganesh A.J. Probabilistic Reliable Dissemination in Large-Scale Systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, March 2003.

[17] V. King and J. Saia. Choosing a Random Peer. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–130, July 2004.

[18] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, and A. Demers. Decentralized schemes for size estimation in large and dynamic groups. In *Proceedings of IEEE Network Computing and Applications (NCA)*, October 2005.

[19] A. McIver and C. Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer Verlag, January 2005.

[20] E. Le Merrer, A.-M. Kermarrec, and L. Massoulie. Peer to peer size estimation in large and dynamic networks: A comparative study. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, June 2006.

[21] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[22] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Style Failure Detection Service. In *Proceedings of Middleware*, September 1998.

[23] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured P2P overlays. *J. Network Syst. Manage.*, 13(2), 2005.

[24] P. Wadler. The Essence of Functional Programming. In *Conference Record of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–14, January 1992.