Intelligent Gossip

Alberto Montresor[1]

University of Trento, Italy
alberto.montresor@unitn.it

Summary. The gossip paradigm made its first appearance in distributed systems in 1987, when it was applied to disseminate updates in replicated databases. Two decades later, gossip-based protocols have gone far beyond dissemination, solving a large and diverse collection of problems. We believe that the story is not over: while gossip is not the panacea for distributed systems, there are still virgin research areas where it could be profitably exploited. In this paper, we briefly discuss a gossip-based "construction set" for distributed systems and we illustrate how intelligent distributed computing could benefit by the application of its building blocks. Simple examples are provided to back up our claim.

1 Introduction to Gossip

Since the seminal paper of Demers et al. [3], the idea of epidemiological (or gossip) algorithms has gained considerable popularity within the distributed systems and algorithms communities.

In a recent workshop on the future of gossip (summarized on a special issue of Operating System Review [12]), there has been a failed attempt to precisely define the concept of gossip. The reason for this failure is twofold: either the proposed definitions were too broad (including almost any message-based protocol ever conceived), or they were too strict (ruling out many interesting gossip solutions, some of them discussed in the next section).

While a formal definition seems out of reach, it is possible to describe a prototypical gossip scheme that seems to entirely cover the intuition behind gossip. The scheme is presented in Figure 1.

In this scheme, nodes regularly exchange information in periodic, pairwise interactions. The protocol can be modeled by means of two separate threads executed at each node: an active one that takes the initiative to communicate, and a passive one accepting incoming exchange requests.

In the active thread, a node periodically (every Δ time units, the *cycle* length) selects a peer node p from the system population through function *selectPeer*(); it extracts a summary of the local state through function

```
1: loop
2:     wait($\Delta$)
3:     $p \leftarrow selectPeer()$
4:     $s = prepareMessage()$
5:     send $\langle \text{REQUEST}, s \rangle$ to $p$
6: end loop
```

(a) active thread

```
1: loop
2:     receive $\langle t, s_p \rangle$ from all
3:     if $t = \text{REQUEST}$ then
4:         $s = prepareMessage()$
5:         send $\langle \text{REPLY}, s \rangle$ to $p$
6:     end if
7:     $update(s_p)$
8: end loop
```

(b) passive thread

Fig. 1. The generic gossip scheme.

prepareMessage(); and finally, it sends this summary to p. This set of operations is repeated forever. The other thread passively waits for incoming messages, replies in case of active requests, and modifies the local state through function *update*().

The scheme is still too generic and can be used to mimic protocols that are not gossip; as an example, we can map the client-server paradigm to this scheme by simply having all nodes selecting the same peer. For this reason, this scheme must be associated with a list of "rules of thumb" to distinguish gossip from non-gossip protocols:

- peer selection must be random, or at least guarantee enough peer diversity
- only local information is available at all nodes
- communication is round-based (periodic)
- transmission and processing capacity per round is limited
- all nodes run the same protocol

These features are intentionally left fuzzy: for example "limited", "local" or "random" is not defined any further.

With this informal introduction behind, we can focus on what makes gossip protocols so "cool" these days. The main reason is robustness: node failures do not cause any major havoc to the system, and can be tolerated in large quantity; message losses often cause just a speed reduction rather than safety issues. Low-cost is another plus: load is equally distributed among all nodes, in a way such that overhead may be reduced to few bytes per second per node.

The cause of such robustness and efficiency can be traced back to the inherently probabilistic nature of gossip protocols. They represent a certain "laid-back" approach, where individual nodes do not take much responsibility for the outcome. Nodes perform a simple set of operations periodically, they are not aware of the state of the entire system, only a very small (constant) proportion of it, and act based on completely local knowledge. Yet, in a probabilistic sense, the system as a whole achieves very high levels of robustness to benign failures and a favorable (typically logarithmic) convergence time.

We claim that if adopted, the gossip approach can open intelligent distributed computing to a whole variety of autonomic and self-* behaviors, bringing robustness to existing intelligent computing techniques. This claim will be backed up in two steps: first, by showing that several important problems in distributed systems have a robust gossip solution; second, by showing how these solutions can be integrated in existing computational intelligence techniques.

2 Gossip Lego: Fundamental Bricks

Beyond the original goal of information dissemination, gossip protocols have been used to solve a diverse collection of problems. More interestingly, it appears now that most of these solutions can be profitably combined to solve more complex problems. All together, these protocols start to look like a construction set, where protocols can be combined as Lego bricks. Figure 2 lists some of these bricks; the following subsections briefly discuss each of them.

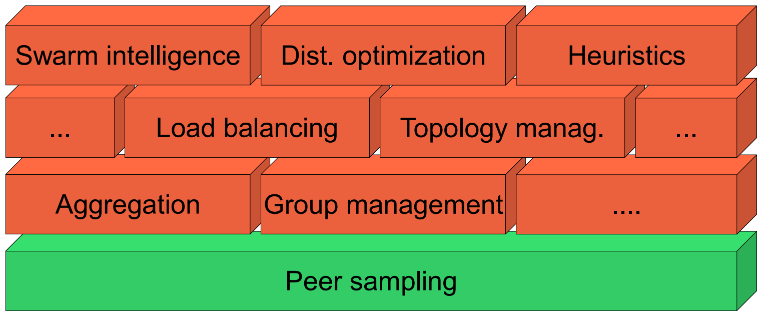

Fig. 2. Gossip Lego: Fundamental bricks

2.1 Peer Sampling

The first piece, *peer sampling*, may be seen as the green Lego baseplate where all kind of models are built. In fact, the problem it solves is at the basis of gossip: how to keep together the population of nodes that constitute the system, in such a way that it is possible implement function $selectPeer()$ that selects nodes from such population.

Instead of providing each node with a global view of the system, the peer sampling service provides each node with continuously up-to-date random samples of the entire population. Higher-level gossip protocols may transparently implement $selectPeer()$ by randomly choosing a per from this sample.

Locally, each node only see the random node returned by *selectPeer*(); globally, the nodes and their samples define an *overlay topology*, i.e. a directed graph superimposed over the network. The graph is characterized by a *random* structure and the presence of a single strongly connected component.

An example instantiation of the peer sampling service is the NEWSCAST protocol [9], characterized by its low cost, extreme robustness and minimal assumptions. The basic idea of NEWSCAST is that each node maintains a local set of random descriptors, called the (partial) *view*. A descriptor is a pair (*node address*, *timestamp*). NEWSCAST is based on the same scheme as all gossip protocols. Function *selectPeer*() returns a random member of the view; function *prepareMessage*() returns the local view, plus a fresh descriptor of itself. Function *update*() keeps a fixed number of freshest descriptors (based on timestamps), selected from those locally available in the view and those contained in the received message. Nodes belonging to the network continuously inject their identifiers in the network with the current timestamp, so old identifiers are gradually removed from the system and are replaced by newer information. This feature allows the protocol to "repair" the overlay topology by forgetting information about crashed neighbors, which by definition cannot inject their identifiers.

Implementations exist in which these messages are small UDP messages containing approximately 20-30 descriptors, each composed of an IP address, a port and a timestamp. The cycle length Δ is typically long, in the range of 10s. The cost is therefore small, few tens of bytes per second, similar to that of heartbeats in many distributed architectures. The protocol provides high quality (i.e., sufficiently random) samples not only during normal operation (with relatively low churn), but also during massive churn and even after catastrophic failures (up to 70% nodes may fail), quickly removing failed nodes from the local views of correct nodes.

2.2 Decentralized Aggregation

Aggregation is a common name for a set of functions that provide a summary of some global system property. In other words, they allow local access to global information in order to simplify the task of controlling, monitoring and optimizing distributed applications. Examples of aggregation functions include network size, total free storage, maximum load, average uptime, location and intensity of hotspots, etc.

An example of gossip-based aggregation algorithm is contained in [8]. The algorithm assumes that each node maintains a local approximation of the value to be aggregated, initially equal to the value of the local property. Function *selectPeer*() exploit the underlying peer-sampling protocol to return a random node. Function *prepareMessage*() returns the current local approximate value, while function *update*() modifies the local approximate value based on some aggregation-specific and strictly local computation based on the previous values. This local pairwise interaction is designed in such a way that all approximate values in the system will quickly converge to the desired aggregate value. For example, in case of average aggregation, at the end of an exchange both nodes install the average of their current local approximate values; after each exchange, the global average will not change, while variance is reduced. It can be proved that at each cycle, the expected reduction is equal to $(2\sqrt{e})^{-1}$, independently of the size of the network.

2.3 Load balancing

The aggregation protocol described above is *proactive*, meaning that all nodes participating in the computation are made aware of the final results. This suggests a simple improvement of a well-known load balancing protocol, as well as showing how simple protocol pieces can be combined together [7].

The load balancing scheme we want to improve works as follows: [1, 13]: given a set of tasks that must be executed by a collection of nodes, nodes periodically exchange tasks in a gossip fashion, trying to balance the load in the same fashion as our average aggregation protocol. The problem with this approach is that tasks may be costly moved from one overloaded node to another overloaded node, without really improving the situation - nodes remain overloaded.

Our idea is based on the concept that moving information about tasks is cheaper than moving tasks. For this reason, we use our aggregation service to compute the average load, and then later we put in contact - through a specialized peer sampling service - nodes that are underloaded with nodes that are overloaded. By avoiding overloaded-to-overloaded exchanges, this algorithm guarantees that an optimal number of transfers are performed.

2.4 Slicing

Once collected all nodes in the same basket through peer sampling, one may want to start to differentiate among them, creating sub-groups of nodes that are assigned to specific tasks. This functionality is provided by a *slicing service*, where the population of nodes is divided into groups (slices) which are maintained, in a decentralized way, in spite of failures.

The composition of slices may be defined based on complex conditions based on both node and slice features; example of possible slice definitions include the following:

- nodes with at least 4GB of RAM;
- not more than 10.000 machines, each of them with ADSL connection or more;
- the group composed by the 10% most performant machines;
- a group of nodes whose free disk space sums up to 1PB.

Several protocols have been devised to solve these problems [4, 6, 14]; all of them are based on special versions of peer sampling. For example, if only nodes with special characteristics (e.g., RAM greater than 4GB) are allowed to insert their node descriptor in exchanged messages, we quickly obtain a sub-population that only contains the desired nodes. By using count aggregation, you can limit the size to a specified value; by ranking values, you can select the top 10%; by using sum aggregation, you can obtain the desired disk space.

2.5 Topology Maintenance

Once you have your slice of nodes, it could be required to organize them in a complex structured topology. T-MAN is a gossip-based protocol scheme for the construction of several kinds of topologies in logarithmic time, with high accuracy [5].

Each node maintains a partial view; as in peer sampling, views are periodically updated through gossip. In a gossip step, a node contacts one of its neighbors, and the two peers exchange a subset of their partial views. Subsequently both participating nodes update their lists of neighbors by merging the the received message.

The difference form peer sampling is how to select peers for a gossip step (function *selectPeer*()), and how to select the subset of neighbors to be sent (function *prepareMessage*()).

In T-MAN, *selectPeer*() and *prepareMessage*() are biased by a *ranking function* that represents an order of preference in the partial views. The ranking function of T-MAN is a generic function and it can capture a wide range of topologies from rings to binary trees, to n-dimensional lattices. For example, in an ordered ring, the preference goes to immediate successors and predecessors over the ring itself. It is possible to demonstrate that several different topologies can be achieved in a logarithmic time.

3 Towards Intelligent Gossip

The bricks presented so far are all dedicated to simple tasks, mostly related to the management of the gossip population itself. You can keep together the entire population though peer sampling, select a group of nodes that satisfies a specific condition through slicing, build a particular topology through T-MAN, and finally monitor the resulting system through aggregation.

But gossip is not limited to this. Recent results suggest a path whereby results from the optimization community might be imported into distributed systems and architected to operate in an autonomous manner. We briefly illustrate some of these results.

3.1 Particle Swarm Optimization

PSO [11] is a nature-inspired method for finding global optima of a function f of continuous variables. Search is performed iteratively updating a small number n of random "particles", whose status information includes the current position vector $\boldsymbol{x}_i$, the current speed vector $\boldsymbol{v}_i$, together with the optimum point $\boldsymbol{p}_i$ and its *fitness* value $f(\boldsymbol{p}_i)$, which is the "best" solution the particle has achieved so far. Another "best" value that is tracked by the particle swarm optimizer is the global best position $\boldsymbol{g}$, i.e. the best fitness value obtained so far by any particle in the population.

After finding the two best values, every particle updates its velocity and position based on the memory of its current position, the best local positions and the best global position; the rationale is to search around positions that have proven to be good solutions, avoiding at the same time that all particles ends up in exactly the same positions.

Nothing prevents the particle swarm to be distributed among a collection of nodes [2]. At each node p, a sub-swarm of size k is maintained; slightly departing from the standard PSO terminology, we say that each swarm of a node p is associated to a *swarm optimum* $\boldsymbol{g}^p$, selected among the particles local optima. Clearly, different nodes may know different swarm optima; we identify the best optimum among all of them with the term *global optimum*, denoted $\boldsymbol{g}$.

Swarms in different nodes are coordinated through gossip as follows: periodically, each node p sends the pair $\langle \boldsymbol{g}^p, f(\boldsymbol{g}^p) \rangle$ to a peer node q, i.e. its current swarm optimum and its evaluation. When q receives such a message, it compares the swarm optimum of p with its local optimum; if $f(\boldsymbol{g}^p) < f(\boldsymbol{g}^q)$, then q updates its swarm optimum with the received optimum ($\boldsymbol{g}^q = \boldsymbol{g}^p$); otherwise, it replies to p by sending $\langle \boldsymbol{g}^q, f(\boldsymbol{g}^q) \rangle$.

Simulations results [2] show that this distributed gossip algorithm is effective in balancing the particles load among nodes; furthermore, the system is characterized by extreme robustness, as the failure of nodes has the only effect of reducing the speed of the system.

3.2 Intelligent heuristics

Many heuristic techniques can be used to approximately solve complex problems in a distributed way. For example, in [10] the problem of placing servers and other sorts of superpeers is considered. The particular goal of this paper is to situate a superpeer close to each client, and create enough superpeers to balance the load. The scheme works in a gossip way, as follows. Nodes randomly take the role of superpeers, and clients are associated to them; then, nodes dissatisfied with the service their receive, start to gossip, trying to elect superpeers that could provide better service. Dissatisfaction could be motivated by the overload of the superpeer, of by the excessive distance between the client and the server. While this heuristics scheme could be stuck in local optima, stills it reasonably improve the overall satisfaction of nodes, especially considering that nodes work in the absence of complete data.

Acknowledgments

Work supported by the project CASCADAS (IST-027807) funded by the FET Program of the European Commission.

References

1. A. Barak and A. Shiloh. A Distributed Load Balancing Policy for a Multicomputer. *Software Practice and Experience*, 15(9):901–913, Sept. 1985.
2. M. Biazzini, A. Montresor, and M. Brunato. Towards a decentralized architecture for optimization. In *Proc. of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, USA, Apr. 2008.
3. A. Demers et al. Epidemic Algorithms for Replicated Database Management. In *Proc. of 6th ACM Symp. on Principles of Dist. Comp. (PODC'87)*, Vancouver, August 1987.
4. A. Fernandez, V. Gramoli, E. Jimenez, A.-M. Kermarrec, and M. Raynal. Distributed slicing in dynamic systems. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*, page 66, Washington, DC, USA, 2007. IEEE Computer Society.
5. M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In S. A. Brueckner, G. Di Marzo Serugendo, D. Hales, and F. Zambonelli, editors, *Engineering Self-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers*, volume 3910 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2006.
6. M. Jelasity and A.-M. Kermarrec. Ordered slicing of very large-scale overlay networks. In *Peer-to-Peer Computing*, pages 117–124, 2006.
7. M. Jelasity, A. Montresor, and O. Babaoglu. A Modular Paradigm for Building Self-Organizing P2P Applications. In *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, number 2977 in Lecture Notes in Artificial Intelligence, pages 265–282. Springer-Verlag, Apr. 2004.
8. M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(1):219–252, 2005.
9. M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, Aug. 2007.
10. G. P. Jesi, A. Montresor, and O. Babaoglu. Proximity-aware superpeer overlay topologies. *IEEE Transactions on Network and Service Management*, 4(2):74–83, Sept. 2007.
11. J. Kennedy and R. Eberhart. Particle swarm optimization. *Neural Networks, 1995. Proceedings., IEEE International Conference on*, 4, 1995.
12. A.-M. Kermarrec and M. van Steen. Gossiping in distributed systems. *Operating Systems Review*, 41(5):2–7, 2007.
13. P. Kok, K. Loh, W. J. Hsu, C. Wentong, and N. Sriskanthan. How Network Topology Affects Dynamic Load Balancing. *IEEE Parallel & Distributed Technology*, 4(3), Sept. 1996.
14. A. Montresor and R. Zandonati. Absolute slicing in peer-to-peer systems. In *Proc. of the 5th International Workshop on Hot Topics in Peer-to-Peer Systems (HotP2P'08)*, Miami, FL, USA, Apr. 2008.