Improving Gossip Dynamics Through Overlapping Replicates

Danilo Pianini[1(✉)], Jacob Beal[2], and Mirko Viroli[1]

[1] Alma Mater Studiorum–Università di Bologna, Cesena, Italy
{danilo.pianini,mirko.viroli}@unibo.it
[2] Raytheon BBN Technologies, Cambridge, USA
jakebeal@bbn.com

Abstract. Gossip protocols are a fast and effective strategy for computing a wide class of aggregate functions involving coordination of large sets of nodes. The monotonic nature of gossip protocols, however, mean that they can typically only adjust their estimate in one direction unless restarted, which disrupts the values being returned. We propose to improve the dynamical performance of gossip by running multiple replicates of a gossip algorithm, overlapping in time. We find that this approach can significantly reduce the error of aggregate function estimates compared to both typical gossip implementations and tree-based estimation functions.

1 Introduction

Gossip protocols are a coordination approach based on estimating a collective state by repeated propagation and aggregation of state estimates between neighboring devices [5,21]. They are widely used in the development of networked and distributed systems, as they can often provide a fast and effective means of enacting strategies for collective adaptation of large numbers of computing devices. This can be particularly important for emerging scenarios and the "internet of things," with the continued rapid increase in both the number of deployed mobile or embedded devices and the networking technologies for connecting them opportunistically. In theory, virtually any collective mechanism—sensing the environment, planning actions, information storage, physical actuation—can be realized by the resilient coordination of large sets of devices deployed in a given region of space [24], and gossip can play an important role as a composable "building block" algorithm for effective programming of such environments [3,22].

Unlike many scenarios where gossip has been deployed and studied, however, in pervasive and embedded environments network connections are typically strongly affected by physical proximity and the effective network diameter may be quite large. Overlay networks, which are often used to ensure that gossip estimates can be rapidly adapted to new circumstances (e.g., [11–13,23]), are often no longer applicable in these circumstances, and we need to find alternate strategies that can enable gossip estimates to adapt rapidly and smoothly to

A. Lluch Lafuente and J. Proença (Eds.): COORDINATION 2016, LNCS 9686, pp. 192–207, 2016.
DOI: 10.1007/978-3-319-39519-7_12

changes in the values being aggregated. We address this challenge by defining a higher-order "time replication" coordination strategy that maintains a set of isolated replicas of a distributed process with staggered start times: applying this strategy to replicate gossip provides a significant improvement over prior approaches as well as an adjustable tradeoff between speed of adaptation and cost of replication.

Design, prototype implementation, and experiments, have been realized by exploiting the toolchain of *aggregate programming* [3], an approach aimed at simplifying the sound engineering of collective adaptive systems by shifting the programming focus from single devices to whole aggregates. This allowed us to smoothly express a formalized version of the proposed approach in terms of the Protelis programing language [20].

Following a brief review of gossip protocols in Sect. 2, we specify the proposed replication strategy in Sect. 3 and analyze the predicted performance of time-replicated gossip in Sect. 4. We then validate these predictions and compare performance against other methods for collective state estimation in Sect. 5, before summarizing contributions and future work in Sect. 6

2 Gossip Protocols

The term *gossip protocol* is used to cover a range of related algorithms and concepts [5,21]. For purposes of this paper, we will formalize gossip with the following generic algorithm, executed periodically on every participating device in unsynchronized rounds:

```
def gossip(f,x) {
  // Declare state variable v, initialized to current value of x
  rep(v <- x) {
    // Every round, merge v with neighbors' values of v and current value of x
    f.apply(x,hood((a,b) -> {f.apply(a,b)},x,nbr(v)));
} }
```

This algorithm begins with an input $x_{\delta,\tau}$ (the input `x`, potentially varying with device δ and time τ) and a fixed merging function f that takes two values of the type of `x` and returns another of the same type. The function f must be idempotent, meaning that $f(a, f(a, b)) = f(a, b)$, and commutative, meaning that $f(a, b) = f(b, a)$. This means that any number of copies of various values of $x_{\delta,\tau}$ can be combined in any order and yet always be guaranteed to eventually produce the same output value $v_{\delta',\tau'}$.

In particular, this algorithm realizes computation of $v_{\delta,\tau}$ by declaring `v` as a state variable (construct `rep`) initialized to $x_{\delta,\tau}$. In every round τ, $v_{\delta,\tau}$ is then updated by using f to combine it with the current value of $x_{\delta,\tau}$ (which may have changed), and with the latest values of $v_{\delta',\tau'}$ that have been shared by the device's current set of neighbors in the network (construct `nbr`, which also implies reciprocally sharing this device's value of $v_{\delta,\tau}$).

For all functions f that are both idempotent and commutative, repeated execution of this gossip algorithm on any connected network with stable inputs ($x_{\delta,\tau} = x_{\delta,\tau'}$) leads to all devices converging to the same value within *diameter* rounds. This algorithm can be optimized in various ways by optimizing the implementation of `rep`, `hood`, and `nbr` (e.g., sharing and computing only on differences), but the essence remains the same.

Gossip is thus a valuable tool for fast, distributed computation of aggregate functions of a network, for any function that can be mapped onto an appropriate f: examples include minimum value, union of sets, and mean value (the last being somewhat more subtle: see [16,21]). By contrast, other approaches to computing a consensus aggregate value are either slow (e.g., Laplacian averaging [9,17]), fragile (e.g., various exact consensus algorithms [10,15], PLD-consensus [1]), or both (e.g., Paxos [6,14]).

The idempotence property of gossip, however, also carries its own significant cost: it is asymmetric and information-destroying. Because a value can be merged in multiple times without affecting the value of the aggregate, it is not possible to know how many times this has actually taken place, and as such there is no inverse function that can be used to remove from the aggregate an input $x_{\delta,\tau}$ that is no longer valid. For example, with $f = \min(a, b)$ values can go down, but they cannot go up again. This means that removing obsolete values from the aggregate function can be difficult and costly. The two main strategies are:

- Values of $x_{\delta,\tau}$ may be in some way time-stamped and/or identified with their source, such that they can be superseded by new information from the same source or discarded if they are not periodically refreshed. Some form of this approach is often used for gossip algorithms that build indexing or routing data structures, such as in peer-to-peer systems (e.g., [11,23]), but has the drawback that either most devices know only a fragment of $v_{\delta,\tau}$ or else that the size of $v_{\delta,\tau}$ and of the updates that need to be shared between neighbors may become very large, since each value of $x_{\delta,\tau}$ needs to be tracked individually.
- The gossip algorithm can be periodically restarted, thus resetting `v` and effectively discarding all old values of $x_{\delta,\tau}$. This has the advantage of being very lightweight but can have significant lags before changes in $x_{\delta,\tau}$ are acknowledged and large transients in $v_{\delta,\tau}$ during the restart. Furthermore, care must be taken to ensure that no values of $v_{\delta,\tau}$ from the old algorithm instance can ever be shared with the new algorithm instance, or else the benefit of restarting will be lost.

In this paper, we focus on the periodic restart strategy, improving its dynamics through a refinement in which multiple overlapping replicates of gossip are run in parallel.

3 Time-Replicated Gossip

The approach we are investigating for improving gossip performance is a simple generalization of the periodic restart strategy for removing obsolete information

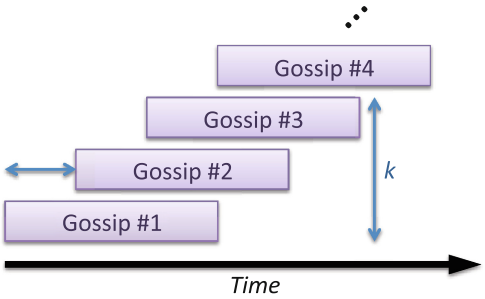


Fig. 1. Time-replicated gossip launches a new gossip process every p seconds, dropping the oldest replicate whenever there are more than k replicates.

from gossip. Rather than maintaining only a single instance of a gossip algorithm, each device will maintain up to k replicates with staggered launch times (Fig. 1). At a period of once every p seconds, a new replicate is launched, and if the full complement of replicates is already running then the oldest replicate will be dropped. This approach provides a compromise solution, avoiding the severe drawbacks of either of the prior methods: the amount of state communicated or stored cannot grow large as there are only k replicates, and large transients in v can be avoided by keeping the current replicate running while new replicates are still stabilizing.

We have implemented this strategy by means of a general time-replication algorithm, coded in Protelis [20] in order to take advantage of the mechanisms of its underlying computational model (the field calculus [7,8]) for succinct encapsulation and manipulation of distributed higher-order functions [3,8]:

```
def timeReplicated(process, default, p, k) {
  rep(state <- [[], 0]) { // [tuples of [replicate, value], oldest replicate ID]
    // Check whether p has elapsed without a new replicate beginning elsewhere
    let newRep = sharedTimer(p,state.get(0));
    // If so, create a new replicate and add it to the collection
    let newProc = if(newRep>0) { [[newRep, default]] } else { [] };
    let procs = state.get(0).mergeAfter(newProc);
    // Execute all processes from self and neighbors, aligning on ID using
    // alignedMap(argument, filter, function to run, default value)
    procs = alignedMap(nbr(processes),
      (replicate, value) -> { replicate >= state.get(1) }, // Ignore old
      (replicate, value) -> { process.apply() }, // Execute process
      default);
    // Prune to keep only the newest k and update the state
    procs = procs.subTupleEnd(max(0, procs.size() - k));
    [procs, procs.map((x) -> {x.get(0)}).fold(min)]
  }.get(0); // Return tuple of [replicate numbers, state] tuples
}
```

In essence, this maintains two pieces of state: the first is a set of running process replicates, each identified by its replicate number, i.e., the first is replicate 1,

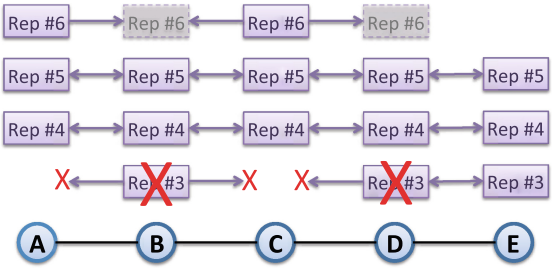


Fig. 2. Example of process replicates created by `timeReplicated`: here four replicates (purple boxes) are running on various subsets of a linear network of five devices (A-E) with $k = 3$, with communication between instances aligned by replicate number via `alignedMap` (purple arrows). Devices A and C have independently started replicate #6, and its instances are spreading new instances to other devices (greyed boxes), merging together as they go. Since $k = 3$, the arrival of replicate #6 also deletes replicate #3 and blocks its spread (red Xs). (Color figure online)

the second replicate 2, etc. The second piece of state is the oldest allowed replicate number, which rises over time as new replicates are created and old ones are discarded.

Every round, each device consults a "shared timer" function to determine whether it should locally launch a new replicate, and if the answer is yes (which happens somewhere in the network at least once every p seconds) then it appends the new replicate with its new, higher identifier, to the end of the current set of processes. This set of replicates are run across the network, using the `alignedMap` primitive to safely encapsulate each replicate, as well as to spread replicates to any other device where such replicates have not already been deemed too old and to merge replicates with other independently launched instances with the same replicate number (Fig. 2).

The `sharedTimer` function is implemented to coordinate with processes spreading via `alignedMap` as follows:

```
def sharedTimer(p,procs) {
  let newReplicate = 0;
  rep(state <- [0,0]) { // [top rep #, time remaining], start rep 1 immediately
    // Compare state replicate to maximum replicate number from elsewhere
    let maxID = max(state.get(0), procs.map((x)->{x.get(0)}).fold(max));
    // When advanced by extension of a process from elsewhere, reset timer.
    if(maxID > state.get(0)) { [maxID, p]
    // When timer expires, signal, advance replicate number, and reset timer.
    } else { if(state.get(1) <= 0) { newReplicate = maxID+1; [maxID+1, p]
    // Otherwise, count down toward timer expiring
    } else { [state.get(0), state.get(1) - self.dt()] }}
  };
  newReplicate // Return zero if nothing changes, otherwise new replicate number
}
```

In essence, this tracks the highest replicate number currently known and the time remaining until a new replicate should be launched. If a spreading process introduces a new replicate, then the replicate number is updated[1] and the timer is reset since a local launch has been pre-empted by an external launch. If, on the other hand, the timer runs out, then this device will launch a new replicate, possibly in parallel with other devices elsewhere.

Thus, a set of distributed timed replicates can be executed without any requirement for synchronization, effectively being launched in either one or many places at the same time, with faster-running devices pulling slower-running devices along after them, i.e., new replicates will tend to be initiated by the device(s) with the fastest clocks. It does not matter where or how many devices launch replicates, since all independent launches with the same number will end up merging in the `alignedMap`.

Whenever the addition of new processes (either locally or by spreading from neighbors) results in there being more than k processes, the oldest are discarded to reduce the number back down to k, and the oldest allowed replicate number updated accordingly. This prevents "old" processes from spreading back into devices where they have already been discarded and ratches the overlapping set of replicates forward incrementally over time.

Ultimately, the replication function returns a tuple of the replicate numbers and values of all currently running replicates. The time-replication algorithm may thus simply be applied to instantiate time-replicated gossip as follows:

```
timeReplicated(() -> gossip(f, x), x, p, k)
```

In other words, it replicates the distributed process $\texttt{gossip}(f, x)$ with process default values taken from x, launch period p and number of replicates k.

In order to apply this approach to improving the dynamics of gossip algorithms, the following questions remain: what are the optimal values for p and k, and how should the values of v returned by each of the different replicates be combined in order to produce the best possible estimate of the true aggregate value v? In the next section, we will address these questions through analysis of the dynamics of gossip.

4 Analysis

The replication approach that we have proposed begins with the intuitive notion that we can avoid large transients and also bound algorithm state by keeping some replicates running while new replicates are started and come to stabilize. Now, let us analyze the process by which replicates launch and stabilize to new values in order to determine how many replicates to create, how frequently to

[1] Note that the algorithm is defined in terms of an unbounded integer; in implementations where there is a desired to use integers with few enough bits to make overflow a realistic possibility, strategies such as lollipop numbering [18] can be used to maintain ordering.

launch new replicates, and how to best make use of the values across multiple replicates.

For this analysis, let us consider an unchanging set of stationary devices, which thus form a fixed network graph of diameter d. Devices execute at the same rate but without any synchronization in phase, sending state updates of all of the values in `nbr` statements to one another once every t seconds. Given the simplicity of the algorithm, we will assume that there is no delay between the time when a device begins to execute a round and the time when its updated values arrive at its neighbors.[2]

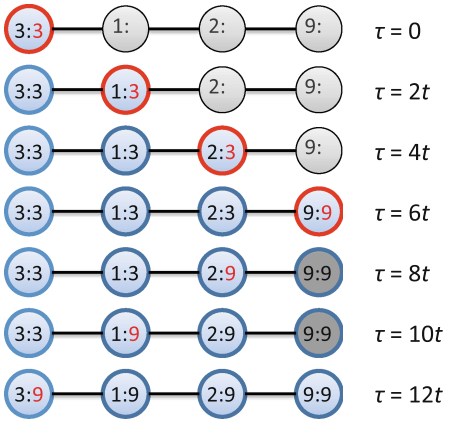


Fig. 3. Illustration of a case in which a new gossip replicate takes the maximum of $4td$ seconds for error from the initial transient to resolve, on a linear network of four devices. Here, gossip is computing the maximum value with $f = max(a, b)$, input x is shown as the left number in each device, and v the right number, which is only instantiated where the replicate is running (blue devices), and not where it has not yet launched (grey devices), and change in each round is indicated in red. The delay is caused by the new replicate launching only at the opposite end of the network from the highest value, and thus the value at this device cannot be correct until the replicate has had time to propagate its launch all the way to the right and for the maximum value to work its way back all the way to the left with, if devices are maximally desynchronized, a delay of $2t$ per hop. (Color figure online)

If devices are perfectly out of synchrony with one another, then it may take up to $2t$ seconds for a message from one device to affect the state of its neighbors. With the right arrangement of states, it may thus take up to $2td$ seconds for gossip replicates to be launched on all devices in the network, with the replicate starting at a single device and spreading to each other device just as its own launch timer expires.

[2] Our analysis may be generalized to devices with drifting clocks and non-trivial execution and transmission time by taking t to be the round length plus execution and transmission delay at the slowest device.

Likewise, the value of a gossip algorithm may have an unboundedly high error at any given device until there has been time enough for information to arrive at that device from every other device, another delay of $2td$ seconds. Consider, for example, gossiping $f = max(a, b)$ when one device has an input of $x = 1000$ and all other devices have value $x = 0$: every device would stay at $v = 0$ until the information from the one $x = 1000$ device reached it, at which point it would instantly leap to $v = 1000$.

Thus, for the first $4td$ seconds after a new gossip replicate begins (i.e., the maximum round-trip time for information to propagate across the network, as Fig. 3 points out), the value of v at any given device may have an unboundedly large "transient" error with respect to its converged value and should not be used.

This gives us a lower bound on when the value computed by a gossip replicate should be used; let us turn now to the opposite side, and consider when the information of a gossip replicate becomes redundant and can be discarded. Consider two sequential replicates of gossip, replicate i and replicate j, where $j = i + 1$. If $x_{\tau',\delta'}$ is the value of x at time τ' and device δ', then for any given device δ at time τ, we can partition the set of all values of $x_{\tau',\delta'}$ into three subsets:

- x_{IJ} are those values used by both replicate i and replicate j.
- x_I are those values used by replicate i but not by replicate j, i.e., those that appeared before replicate j launched.[3]
- x_0 are those values used by neither replicate.

Because the gossip function f is idempotent and commutative, we can thus reorganize the computation of the output values of the two replicates as:

$$v_{\tau,\delta,i} = f(f(x_I), f(x_{IJ})) \tag{1}$$

$$v_{\tau,\delta,j} = f(x_{IJ}) \tag{2}$$

abusing notation to consider $f(X)$ as f applied in arbitrary order to combine all members of set X. By the idempotence and property of f, it must be the case that $v_{\tau,\delta,i} = v_{\tau,\delta,j}$ unless there are values in x_I that are not in x_{IJ}.

Thus, we have that, once replicate j is past its initial transient (i.e., every device has been affected by the value of every other device) the outputs of replicate i and replicate j must be identical, except in the case where the value of replicate i is being affected by input values of $x_{\tau',\delta'}$ from before the launch of replicate j at δ'. Since ignoring such "obsolete" values is the entire point of replication, we thus see that as soon as a replicate has passed its initial transient, there is no reason to consider the output of any older replicate: the older replicate must be either identical or obsolete.

From these deductions, we now have answers to two of our questions about replication. First, only the output value v of the oldest replicate should be used.

[3] Note that no values can be used by replicate j but not replicate i, because replicate j cannot be launched on any device before replicate i is also launched at that device.

Second, replicates should be retained only until the next replicate has stabilized, at which point they are obsolete and may be discarded. More precisely, we may state this relationship in the form of an equation:

$$p = \frac{4dt}{(k-1)} \tag{3}$$

In other words, with one replicate providing the current "safe" estimate for v and $k-1$ later replicates maturing, with each replicate taking up to $4dt$ seconds to mature, replicated gossip can sustain a steady state in which one replicate matures every $\frac{4dt}{(k-1)}$ seconds.

What remains is the question of the size of k, or conversely of p. Unlike the other relations that we have considered, however, there is no optimal choice here, but rather a tradeoff between the speed with which obsolete information can be removed from a gossip and the number of replicates being maintained (with accompanying requirements for communication, computing, and memory resources). Thus, once a choice has been made for either k or p, the optimal value for the other parameter can be determined with the aid of any conservative estimate of the diameter of the network. Prioritizing the number of replicates, as diameter may often change dynamically over time, we can implement replicated gossip in Protelis as follows:

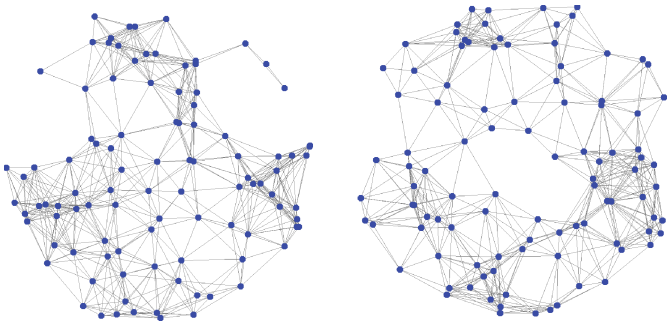
```
def tr_gossip(f, x, k, d) {
  // Compute p by Eq. 3
  let p = 4 * d * self.dt() / (k - 1);
  // Run replicated gossip and return the value from the oldest replicate (first tuple)
  timeReplicated(() -> gossip(f, x), x, p, k).get(0).get(1).
}
```

In terms of managing the tradeoff between number of replicates and speed of adaptation, from Eq. 3, we can see that the duration a replicate will persist (and thus potentially obsolete gossip inputs as well) will be $\frac{k+1}{k} \cdot 4dt$. Thus, if the minimum of two replicates is used, then obsolete information can persist for up to $8dt$ seconds, while if the number of replicates is allowed to grow without bound, the minimum time for obsolete information to persist is $4dt$. In between, a small handful of replicates is likely all that is necessary to get to a point where the diminishing returns on adaptation speed are not worth the additional cost in communication.

In practice, the better the estimate of diameter, the closer to optimal the tradeoff between adaptation speed and size can be made. Likewise, improvements in synchronization guarantees between devices may reduce the conservative $4dt$ closer toward the theoretical minimum of dt.

5 Experimental Validation of Performance

We now validate the performance of time-replicated gossip in simulation, comparing the performance for three representative gossip functions against several



(a) $\tau = 0$ (b) $\tau = 20$

Fig. 4. Simulations are run on a unit disc graph over devices moving via Lévy walks from a random initial distribution: (a) shows a typical initial network and (b) the modified network after 20 rounds of simulation.

prior methods. All experiments have been performed using the Alchemist simulator [19] and Protelis algorithm implementations[4].

5.1 Experimental Setup

For our experiments, we compare the computation of three gossip functions, chosen as representative typical applications of gossip: The three gossip functions are minimum ($f = \min(a, b)$), logical AND ($f = and(a, b)$), and estimated mean (using the method presented in [16][5]). We compared the execution of time-replicated gossip (defaulting to $k = 5$) on these gossip functions with four representative prior methods for estimating aggregate functions, two gossip and two non-gossip:

1. **Gossip:** the baseline algorithm, as defined in Sect. 2, and never restarted.
2. **R-Gossip:** gossip restarted periodically, as discussed in Sect. 2, implemented by time-replicated gossip with $k = 1$.
3. **C+G:** estimate is computed over a spanning tree, then broadcast from the root; the name is taken from the particular implementation we use, which combines the `C` and `G` "aggregate building blocks" from [4].
4. **Laplacian-based consensus (mean only):** incrementally estimates mean $x_{\delta,\tau}$ by in each round adding to the current estimate α times the difference with the neighbor's estimates and the current $x_{\delta,\tau}$ (using $\alpha = 0.04$, which is expected to be fast yet stable per [17]).

All algorithms are executed in parallel on a simulated network of n devices distributed within a circular arena, each device uniquely identified by numbers

[4] Full code at: https://bitbucket.org/danysk/experiment-2016-coordination.

[5] Note that this method uses random numbers, so in order to ensure that replicates are identical except when given different inputs, we seed the pseudorandom generators identically for all replicates on a given device.

0 to $n-1$. Devices execute unsynchronized, with random phase but at the same rate $t = 1$. Devices communicate with all other devices within 1 unit distance, and the radius of the arena is chosen as $\sqrt{\frac{n}{m}}$, such that every device will have an expected m neighbors. In particular, we use $m = 15$ neighbors, a value that ensures the network is mostly well-connected and that $d = 2\sqrt{\frac{n}{m}}$ is a reasonable estimate of its diameter. Initial positions are selected uniformly randomly, and thereafter devices move randomly within the circular arena following a speed s reactive Lévy walk [2]. Figure 4 shows snapshots of an initial deployment and its evolution. Except where otherwise noted, simulations use $n = 100$, giving an estimated diameter of just over 5 hops and $s = 0.05$, meaning that a device is expected to move a length equal to the diameter of the arena in a little over 100 rounds, and for each condition use 40 simulation runs of 300 rounds each.

Our experiments challenge adaptation dynamics by using a set of input values $x_{\delta,\tau}$ that are spatially correlated and have two large discontinuous changes across both space and time. For the mean and minimum functions $x_{\delta,\tau}$ is defined as:

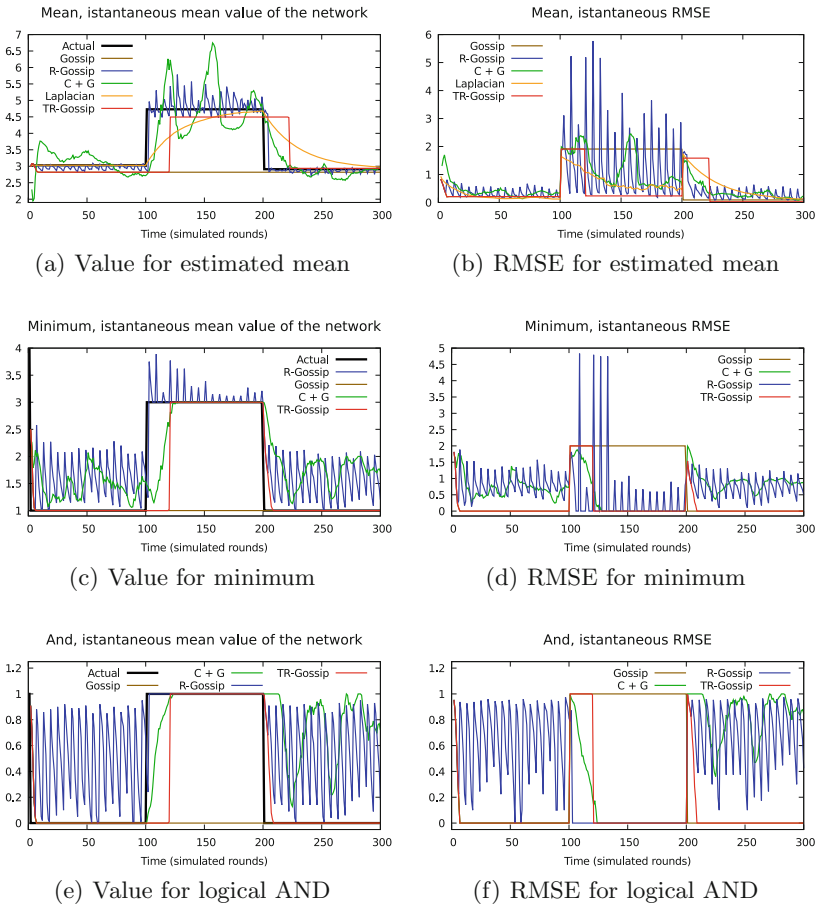
- $\tau < 100$: Devices on the left half of the arena at $\tau = 0$ have $x_{\delta,\tau} = 2$, while those on the right have $x_{\delta,\tau} = 4$, except device 1 has $x_{\delta,\tau} = 1$
- $100 \leq \tau < 200$: Devices on the left half of the arena at $\tau = 100$ have $x_{\delta,\tau} = 6$, while those on the right have $x_{\delta,\tau} = 3$, except device 1 has $x_{\delta,\tau} = 50$
- $200 \leq \tau$: Devices on the left half of the arena at $\tau = 200$ have $x_{\delta,\tau} = 2$, while those on the right have $x_{\delta,\tau} = 4$, except device 1 has $x_{\delta,\tau} = 1$

For logical AND, all devices have $x_{\delta,\tau} =$ true except that device 0 is false for the first 100 rounds and device 1 is false for the final 100 rounds.

5.2 Convergence Dynamics

First, we examine a single simulation run in order to compare in detail the dynamics by which time-replicated gossip converges to a correct value against the convergence dynamics of the alternative algorithms. Figure 5 shows the evolution of mean value and mean root mean squared error (RMSE) across all devices for each function. As predicted, time-replicated gossip is safe from any unexpected transients, tracking to the correct value after a short delay. When the new value follows the monotonic direction of the function this is very fast (as in the second transition for minimum and logical AND); otherwise is must wait the full pk delay for all affected replicates to be discarded.

Non-restarted gossip, by contrast, can never discard old information, and thus cannot adapt during the 100–200 time interval, while non-replicated restarting gossip adapts quickly but experiences periodic sharp error transients at every restart. Laplacian consensus exhibits a smooth but very slow convergence, since values are not homogeneously distributed [9], and error never reaches zero, indicating that the apparently good mean value actually represents not correct values but balanced distribution of overestimates and underestimates. Finally, C+G continuously tries to converge to the correct value, but its tree structure continually gets disrupted by changes in the network structure, and therefore it shows a strong and variable error throughout the whole experiment.



(a) Value for estimated mean (b) RMSE for estimated mean

(c) Value for minimum (d) RMSE for minimum

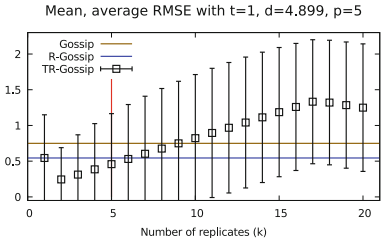
(e) Value for logical AND (f) RMSE for logical AND

Fig. 5. Evolution of mean value across devices (a,c,e) and root mean squared error (RMSE) (b,d,f) for the three functions under test: estimated mean (a,b), minimum (c,d), and logical AND (e,f).

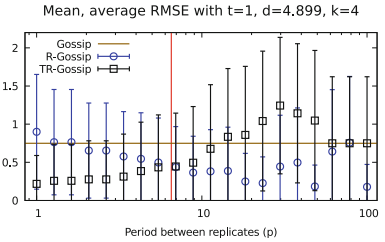
5.3 Effect of Varying of k and p

Our analysis in Sect. 4 identified an optimal conservative relationship between number of replicates k and replicate period p given a particular network diameter d. In practice, however, network diameter may frequently change and can be costly or difficult to estimate precisely, so it is important that estimation not be badly effected by the use of suboptimal parameters.

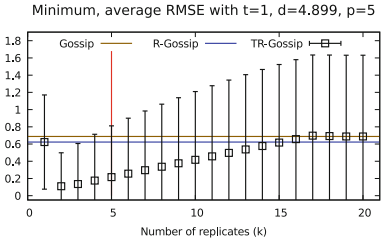
As the analysis was quite conservative, we should expect that as the duration covered by replicates is reduced (e.g., by reducing either k or p while holding the other fixed), error should gradually decrease as the delay to adapt is reduced. At some point, however, the transients of new replicates will not have had time to resolve and error will increase. Complementarily, increasing the duration covered



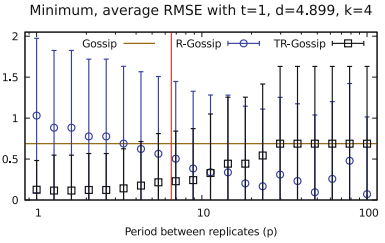
(a) Varying k for estimated mean



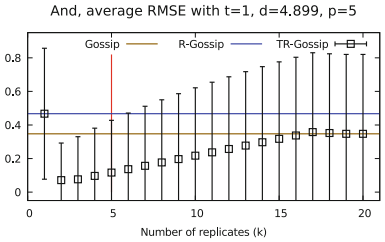
(b) Varying p for estimated mean



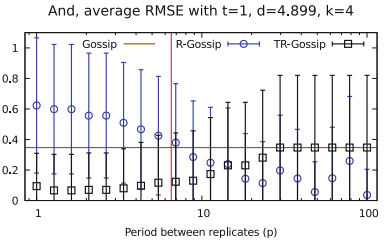
(c) Varying k for minimum



(d) Varying p for minimum



(e) Varying k for logical AND

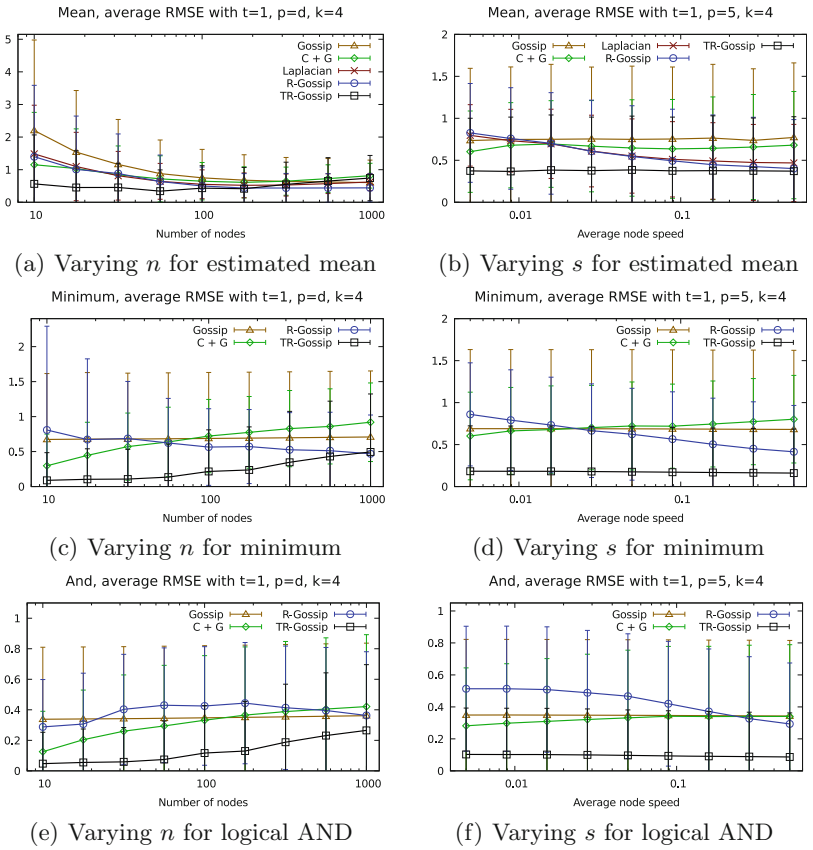


(f) Varying p for logical AND

Fig. 6. Effect of varying number of replicates k (a,c,e) and replication period p (b,d,f) on mean RMSE. The red vertical line marks the value (for k and p respectively) that is suggested by our analysis. Non-restarted gossip and single-replicate restarting gossip (with restart time equal to p) are plotted for comparison. Error bars indicate the standard deviation of the average RMSE across the 40 simulation runs. (Color figure online)

by replicates will not expose transients but will increase error incrementally as the delay to adapt increases.

Figure 6 shows the results of testing these hypotheses against k varying from 1 to 20 and p varying geometrically from 1 to 100. As predicted our analysis is shown to be quite conservative: error in fact decreases with decreasing k and p until the very smallest values. Likewise, it increases smoothly with increasing k or p until it is so high that it saturates the experimental conditions and in some cases actually begins to decrease due to aliasing. As such, it appears that in practice the values for p can indeed be set significantly more aggressively than the bound computed in Sect. 4.



(a) Varying n for estimated mean

(b) Varying s for estimated mean

(c) Varying n for minimum

(d) Varying s for minimum

(e) Varying n for logical AND

(f) Varying s for logical AND

Fig. 7. Resilience of time-replicated gossip to changes in network size and volatility: its mean RMSE over time is not significantly degraded by varying number of devices n (a,c,e) or speed s (b,d,f), while other algorithms perform worse except under extreme conditions. Error bars indicate the standard deviation of the average RMSE across the 40 simulation runs.

Paradoxically, restarting gossip actually improves its performance as p increases, but due to the fact that less frequent restarts mean that its values are less often disrupted by transients. Thus, when the values of k and p are far from optimal, the mean error of replicated gossip is worse than restarting gossip and occasional transients may actually be less disruptive than overly long delays waiting for adaptation, depending on application.

5.4 Resilience to Network Size and Volatility

Finally, we tested how well time-replicated gossip scales to larger networks and adapts to differences in network volatility by changing the number of devices n and speed s. For each parameter, we evaluated a geometric distribution of nine

values across two orders of magnitude, ranging n from 10 to 1000 and s from 0.05 to 0.5, respectively. Results are shown in Fig. 7.

Since increasing the number of devices increases the diameter in our experiments, the time-replicated gossip should degrade incrementally due to the increased time before old replicates can be safely discarded, and indeed this is what is observed. In larger networks, therefore, the mean advantage of time-replicated gossip over other approaches decreases, and in fact in the conditions we evaluate it is slightly outperformed by the faster but more volatile methods for estimated mean. In some circumstances, however, delay may still be preferable to unpredictable transients.

Higher speed of devices is expected to affect the network by decreasing its effective diameter but increasing the frequency of topology changes. Neither of these should affect time-replicated gossip, given its conservative diameter estimate, and indeed speed appears to have no significant effect on its performance. Single-replicate restarted gossip and Laplacian averaging, on the other hand, benefit greatly from a reduced effective diameter that decreases the transients they suffer, while C+G performs worse as the amount of topological disruption increases.

6 Contributions and Future Work

In this paper, we have introduced a time-replication method that significantly improves the dynamical performance of gossip-based distributed state estimation. Analysis bounds the time to maintain replicates by the round-trip time of information across the network and identifies an adjustable tradeoff between improved performance and number of replicates, and these conclusions are validated by experiments in simulation.

Future work can further improve performance by enabling tighter self-adjustment of parameters. In particular, a network diameter estimation algorithm, improved synchronization, and monitoring of transient length can all be employed to decrease the required replication interval, thereby allowing faster adaptation. Second, time-replicated gossip can be applied to any number of systems in which gossip is being used, in order to improve their performance. Finally, the generic nature of the time-replication algorithm we have introduced makes it a candidate for future studies to evaluate if and how time-replication can be used to improve other classes of distributed algorithms.

References

1. Beal, J.: Accelerating approximate consensus with self-organizing overlays. In: Spatial Computing Workshop, May 2013
2. Beal, J.: Superdiffusive dispersion and mixing of swarms. ACM Trans. Auton. Adapt. Syst. **10**(2), 1–24 (2015)
3. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. IEEE Comput. **48**(9), 22–30 (2015)

4. Beal, J., Viroli, M.: Building blocks for aggregate programming of self-organising applications. In: IEEE SASO Workshops, pp. 8–13 (2014)
5. Birman, K.: The promise, and limitations, of gossip protocols. ACM SIGOPS Oper. Syst. Rev. **41**(5), 8–13 (2007)
6. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Principles of Distributed Computing, pp. 398–407 (2007)
7. Damiani, F., Viroli, M., Beal, J.: A type-sound calculus of computational fields. Sci. Comput. Program. **117**, 17–44 (2016)
8. Damiani, F., Viroli, M., Pianini, D., Beal, J.: Code mobility meets self-organisation: a higher-order calculus of computational fields. In: Graf, S., Viswanathan, M. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. LNCS, vol. 9039, pp. 113–128. Springer, Heidelberg (2015)
9. Elhage, N., Beal, J.: Laplacian-based consensus on spatial computers. In: International Conference on Autonomous Agents and Multiagent Systems, pp. 907–914 (2010)
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM (JACM) **32**(2), 374–382 (1985)
11. Gupta, I., Birman, K., Linga, P., Demers, A., van Renesse, R.: Kelips: building an efficient and stable P2P DHT through increased memory and background overhead. In: Kaashoek, M.F., Stoica, I. (eds.) Peer-to-Peer Systems II. LNCS, vol. 2735, pp. 160–169. Springer, Heidelberg (2003)
12. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Trans. Comput. Syst. (TOCS) **23**(3), 219–252 (2005)
13. Jelasity, M., Montresor, A., Babaoglu, O.: T-man: gossip-based fast overlay topology construction. Comput. Netw. **53**(13), 2321–2339 (2009)
14. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
15. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
16. Mosk-Aoyama, D., Shah, D.: Fast distributed algorithms for computing separable functions. IEEE Trans. Inf. Theor. **54**(7), 2997–3007 (2008)
17. Olfati-Saber, R., Fax, J.A., Murray, R.M.: Consensus and cooperation in networked multi-agent systems. Proc. IEEE **95**(1), 215–233 (2007)
18. Perlman, R.J.: Fault-tolerant broadcast of routing information. Comput. Netw. **7**, 395–405 (1983). http://dx.org/10.1016/0376-5075(83)90034-X
19. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simul. **7**(3), 202–215 (2013)
20. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: ACM Symposium on Applied Computing, pp. 1846–1853 (2015)
21. Shah, D.: Gossip Algorithms. Now Publishers Inc, Norwell (2009)
22. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organising systems by self-stabilising fields. In: IEEE SASO, pp. 81–90 (2015)
23. Voulgaris, S., van Steen, M.: An epidemic protocol for managing routing tables in very large peer-to-peer networks. In: Brunner, M., Keller, A. (eds.) DSOM 2003. LNCS, vol. 2867, pp. 41–54. Springer, Heidelberg (2003)
24. Zambonelli, F.: Toward sociotechnical urban superorganisms. IEEE Comput. **45**(8), 76–78 (2012)