Decentralized Management of Random Walks over a Mobile Phone Network

Árpád Berta
University of Szeged, Hungary
Email: berta@inf.u-szeged.hu

Márk Jelasity
University of Szeged, Hungary
Email: jelasity@inf.u-szeged.hu

***Abstract*—Gossip learning is a form of decentralized stochastic gradient descent search that is implemented through randomized walks within a network. Our goal is to enable one to deploy gossip learning in open distributed systems, for example, in overlay networks formed by mobile devices, where different data mining tasks could be launched by many users. Among the many problems this long term goal raises, here we focus on the problem of running many random walks simultaneously. This is a challenging problem in itself in a decentralized setting because all the walks have to be persistent (they have to perform many hops) and agile (they need to move quickly). At the same time, the solution must take hard bandwidth constraints into account. Here, we propose a protocol to manage $O(n)$ random walks in a network of n nodes. Although our motivation is gossip learning, this protocol may be viewed as a general middleware service for the management of walks over networks. A key element of our protocol is a multi-level restarting mechanism designed to prevent the failure of random walks due to node churn, while respecting a set of bandwidth constraints. Here, we simulate our solution using a trace collected from real smartphones. We demonstrate that the random walks are kept alive and are run at close to optimal speed under the given bandwidth constraints.**

***Keywords*-decentralized data mining, gossip, churn, random walks, mobile phone networks**

I. Introduction

Gossip learning [1] is a decentralized approach to machine learning that is based on stochastic gradient descent search. Here, the model that is being fit on the data performs a uniform random walk over the network and it is updated before each step using the local data. Recently, the same idea has been applied to matrix factorization as well that is useful, for example, in implementing decentralized recommender systems [2].

Our long-term research goal is to allow gossip learning to be deployed in a multi-user decentralized environment where users or software agents can launch learning tasks over the collection of the local data of the participants of the network. We are interested in multi-user environments as our goal is to create a fully open collaborative environment where those who provide data can also enjoy the benefit of mining the collective data of the community. The notion of decentralization is also important, as has been recognized by other researchers as well. One reason is that distributed computing allows better scalability compared to cloud-based solutions by exploiting local resources and networks, as proposed e.g. by Cisco in its ongoing fog computing initiative [3]. Another reason is the increasing need for privacy as the personal data collected and stored by ubiquitous personal computing devices such as smart meters, sensors and mobile devices is becoming richer and richer [4].

Multi-user gossip learning raises many research problems. Among those, here we focus on the management of multiple random walks. As we describe more precisely later, we assume that in an overlay network many random walks are run, each representing a learning task. Each task might be owned by a separate user. Our problem here is to ensure that all the walks keep progressing in spite of benign faults caused by nodes joining and leaving (that is, node churn). Also, we want all the walks to progress quickly, that is, without delay, taking the bandwidth into account that is assigned to the nodes.

It should be mentioned here that—although our motivation is gossip learning for multi-user environments—the random walk management middleware service is potentially useful in a more general context as well as random walks have been applied to many different functions other than machine learning. In early peer-to-peer systems they were proposed to implement search [5], and since then random walk techniques have been applied to various functions that include a membership service [6]–[8], sampling [9], and for computing various queries including community detection in large networks [10] and network size [6], just to name a few.

The challenge lies in the decentralized nature of the system. Our solution is based on a multi-level protocol in which we have three "lines of defense" that are similar to competence levels in a hierarchical organization. Problem solving is first attempted at the lowest level and in the case of failure the problem is escalated to the next level. The first two levels are completely decentralized. Ideally, these two decentralized mechanisms should handle the vast majority of faults and the third level—which is implemented by the central control of the owner—should be reached only very rarely. The motivation behind this design philosophy is that we wish to offer a conceptually simple and cheap solution that avoids accessing central resources almost all the time, as opposed to a complex and/or expensive protocol that provably works without any central control all the time.

Our contribution is twofold. First, we propose a multi-level decentralized protocol to run $O(n)$ random walks in a network of n nodes that can tolerate benign failures. Second, we demonstrate through simulation that the protocol indeed protects the random walks and that the walks progress at a near-optimal speed. We base our simulations on a real

In: Proc. PDP 2017, pp 100–107, http://dx.doi.org/10.1109/PDP.2017.73.

trace of smart phones we collected [11]. Although we focus on collaborative mobile platforms [12], multi-user gossip learning applications may be found in smart metering [13] and over Internet of Things platforms [14] as well.

II. System Model

We model our system as a large set of nodes that communicate via message passing. We assume a reliable transfer protocol. This implies that messages are not dropped, so communication fails only if the source or target node fails before transferring the full message. At every point in time each node has a set of neighbors. The neighbor set can change over time, but nodes can send messages only to their current neighbors. We will assume here that the set of neighbors is a uniform random sample from the network (see, for example, [8]). Nodes can leave the network or fail at any time. In our simulations we will assume that when a node leaves the network it retains a subset of its state until it joins the network again, but this is not a critical assumption. Messages can be delayed up to a finite delay. We do not assume synchronized time.

III. The Multiple Random Walk Service

Let us first define random walks at the level of abstraction that is required for the description of our algorithms. A random walk may be viewed as a mobile agent with a state (consisting of payload and metadata) that jumps from node to node at random. The nature of random neighbor selection is not critical here, but it does affect load balancing, so here we assume a random node is picked from the network with the help of a suitable peer sampling service. The payload of the walk is application dependent. In gossip learning, it represents a machine learning model that is updated at each node based on local information.

The metadata of the random walk includes a unique walk ID, a restart ID unique within the scope of the same walk ID, and a step count that counts the hops completed by the walk. When the node responsible for the current hop fails before successfully completing the hop, the walk will be restarted using an earlier state that is (hopefully) still available in some previously visited nodes. The restarted walk will have the same walk ID, but it will get a new restart ID.

In the systems we envision there will be n nodes and $O(n)$ random walks each working on different tasks. The problem we wish to solve is to provide a fault tolerant implementation that is able to restart the failed walks without creating redundant copies.

A. Bird's Eye View

In a nutshell, our solution is made up of three conceptual levels. At the first (lowest) level a local mechanism is implemented. Here, after completing a random walk hop, every node monitors the success of the next hop. In the case of a failure, the monitoring node will restart the walk. This mechanism is local because the monitoring node retains a copy of the payload that it has just transmitted to the monitored node.

The idea is that in the vast majority of failures this local mechanism will fix the problem, but when it does not, the problem gets escalated to level two. This happens when the monitoring node fails before it can detect the failure of the walk. The node performing the current hop therefore monitors the monitoring node (called the supervisor) and invites a new supervisor if the current supervisor fails. This new supervisor, however, might not store the payload, or it might store only an outdated copy, so at level two a more expensive mechanism has to be used. Namely, when detecting a failure, the supervisor broadcasts the restarting request that will eventually reach those nodes that have fresh versions of the payload. These nodes then attempt to restart the walk in a sequential order determined by how old their copy of the payload is. After a successful restart another broadcast is sent about the success, which prevents further attempts at restarting. A simple mechanisms is also in place to stop most of the redundant restarted walks.

The third (and final) level is implemented by the central control carried out by the owner of the walk. The random walk can report its state to the owner regularly (if the owner is reachable), which allows for appropriate interventions. In our simulations this happens extremely rarely, as we will demonstrate later on.

As can be seen from this short summary, we opt for a best effort multi-level mechanism without formal guarantees that nevertheless attempts to escalate as little work as possible to the increasingly expensive upper levels. We believe that in the complex systems we focus on this is a preferable approach that allows us to carry out most of the control tasks in a decentralized way while keeping the system conceptually simple and manageable.

B. Detailed Description of the Protocol

The pseudo code of the protocol run by all the nodes can be seen in Algorithm 1. As for the local state of the node, *sendQueue* is a FIFO queue that keeps sending its next entry to a random node until the queue is not empty. If the recipient node fails before completing the transaction, the queue selects another random node and tries sending the current entry again until the transmission succeeds. This queue also informs the node about each successful transmission by invoking the method onTransmissionComplete(). In this method we simply cancel the monitoring (supervision) of this completed hop and start to monitor the next hop. We also store the walk in *storageQueue*, which is also a FIFO queue with a fixed storage capacity. When it is full, the next entry is removed.

When a random walk arrives successfully, onRandomWalkArrival() is invoked where the node records which node its level one supervisor is, then the payload is updated and the next hop is scheduled.

Failure detection is implemented via the event handler onConnectionTimeout() that is invoked when a neighbor that the node is currently in contact with fails. Here, if a monitored node fails then in the case of level one monitoring (the node was the previous sender) the node simply schedules the restarting of the walk while also assigning a supervisor to itself. In the case of level two monitoring (the node does not have the (fresh) payload) the node schedules for broadcast a new request for restarting the walk. Finally, if the failing node was the node's supervisor then a new one is selected. Note that we do not detail the algorithm for finding (or replacing) supervisors here; it involves contacting live nodes

Algorithm 1 Multiple Random Walk Protocol

```
1: δ:                                   ▷ estimated time for full broadcast
2: Δ:                                   ▷ gossip round length
3: sendQueue:                           ▷ queue where walks to be forwarded wait
4: storageQueue:                        ▷ queue where we store recent random walks
5: rwEvents:                            ▷ fresh events broadcast to manage random walks

6: loop                                 ▷ push-pull gossip protocol to broadcast walk events
7:     wait(Δ)
8:     p ← selectPeer()
9:     rwEvents.cleanup()
10:    send rwEvents to p
11:    send pull request to p

12: procedure ONRECEIVERWEVENTS(rwEvents')
13:    for event in rwEvents' \ rwEvents do   ▷ examine the new events
14:        if rwEvents.isObsolete(event) then
15:            continue                        ▷ jump to next event
16:        if event type is RestartRequest then
17:            if storageQueue.contains(event.rw) then
18:                restartThreadsFactory.start(event)
19:        else if event type is Restarted then
20:            restartThreadsFactory.stop(event)
21:            if rwEevents.containsConflict(event) then
22:                event ← new MultipleRestarts(rwEvents,event)
23:        if event type is MultipleRestarts then
24:            restartThreadsFactory.stop(event)
25:            for rw in sendQueue do
26:                if conflict(rw,event.rw) then     ▷ kill redundant walk
27:                    sendQueue.remove(rw)
28:                    p ← getSupervisor(rw)         ▷ either level 1 or 2
29:                    send cancelSupervision(rw) to p
30:        rwEvents.add(event)
31:        rwEvents.cleanup()

32: procedure RESTARTTHREADSFACTORY.START(event)
33:        ▷ this should be run in a new thread, presentation is simplified
34:    rw ← storageQueue.get(event.rw)
35:    window ← event.rw.steps − rw.steps
36:    restartWindowStart ← window·δ+ event.creationTime()
37:    restartWindowEnd ← (window+1)·δ+ event.creationTime()
38:    wait while currentTime() < restartWindowStart
39:    if currentTime() < restartWindowEnd then
40:        sendQueue.add(rw)                         ▷ level 2 restart
41:        supervisorAtLevel2.add(newSupervisor(rw))
42:        rwEvents.add(new Restarted(rw))

43: procedure ONCONNECTIONTIMEOUT(p)
44:    for rw in getSupervisedRWsAtLevel1(p) do
45:        sendQueue.add(rw)                         ▷ level 1 restart
46:        supervisorAtLevel2.add(newSupervisor(rw))
47:    for rw in getSupervisedRWsAtLevel2(p) do
48:        rwEvents.add(new RestartRequest(rw))
49:    if isSupervisor(p) then                       ▷ either level 1 or 2
50:        supervisorAtLevel2.add(replaceSupervisor(p))

51: procedure ONRANDOMWALKARRIVAL(rw, p)
52:    supervisorAtLevel1.add(rw, p)
53:    update(rw)
54:    sendQueue.add(rw)

55: procedure ONTRANSMISSIONCOMPLETE(rw, p)
56:    q ← getSupervisor(rw)                         ▷ either level 1 or 2
57:    send cancelSupervision(rw) to q
58:    supervisedAtLevel1.add(rw, p)
59:    storageQueue.add(rw)
```

from the network and negotiating with them. Note also that the failing node might have been the supervisor for more than one walk at both level one and two, so all instances need to be replaced.

Let us now move on to the discussion of the second level where restarting is achieved through various broadcast messages. To implement the broadcast primitive, each node runs a basic push-pull gossip broadcast protocol in an active loop with round length Δ. The local set *rwEvents* contains those messages that are currently actively broadcast. Each message is gossiped up to a given maximal number of hops that is set such that all the nodes receive the broadcast with very high probability. The method rwEvents.cleanup() removes those messages that have reached this limit.

There are three kinds of gossip messages, namely RestartRequest, Restarted and MultipleRestarts. All of these messages refer to the failure of a given random walk instance, identified by the walk ID and the restart ID. For this reason, from now on we will assume that the messages mentioned in the discussion belong to the same failure event, unless otherwise stated.

A RestartRequest is generated by a level two supervisor when it detects that a walk has failed. This request has a reference to the walk ID and the restart ID that failed. A Restarted message is generated by a node when it decides to restart a walk based on a RestartRequest. Apart from the walk ID and the old restart ID, this event also refers to the restart ID of the new walk. A MultipleRestarts message is generated by a node that receives multiple Restarted messages that belong to independent restarts of the same walk following the same failure event. Once again, this event refers to the walk ID and the old restart ID, and in addition it contains the new restart ID that is picked to be kept alive.

Method onReceiveRWEvents() processes the incoming broadcast messages. There, only the new messages are processed that are not already included in the local set. First it is tested whether a given message is obsolete or not. This is defined based on a natural dominance relation over the messages. Restarted messages dominate RestartRequests and MultipleRestarts messages dominate the other two types.

In addition, within a given type, an older RestartRequest (with the larger step count) dominates a younger one. Note that normally there should be only one RestartRequest being broadcast but due to the unreliability of the applied failure detector we could in theory have more than one active supervisor for the same walk so more requests may get generated. Messages of type Restarted do not dominate each other, instead, multiple Restarted messages indicate a failure (redundant restarts). In this case a MultipleRestarts message is generated that contains information about which new walk to keep alive: this will be the one with the minimal restart ID. MultipleRestarts messages also have a dominance relation: the message with the smaller restart ID to keep alive wins. Note that due to the timing variance and unreliability of the broadcast primitive, different conflicts might be picked up by different nodes so we may indeed have various different MultipleRestarts messages.

The non-dominated new messages are then processed. In the case of a RestartRequest event if the node has a copy of the payload then a restart timer is started in a separate timer

thread. This thread calculates a restart window in which this node is allowed to restart the walk. The window depends on the age of the local copy of the payload. This way, the different copies of the payload in the network attempt a restart in a sequential order with a high probability, avoiding redundant copies. The window is relative to the first creation of the RestartRequest. Knowing the creation time of the request does not necessarily require synchronized clocks, as an approximation is sufficient that can be computed via summing the approximate hop-times during broadcast.

When a Restarted event is received, we stop any restart timers for this walk and check for conflicting (that is, redundant) restarts. Should there be any such redundant restarts, a new MultipleRestarts event is placed in the broadcast message set.

Finally, when a MultipleRestarts event arrives, the node stops any related restart timers and it also removes all the copies of the redundant walk while also canceling any supervisors for these walks. The *sendQueue* also checks *rwEvents* before sending the next message for possible conflicts (not indicated in the pseudo code).

The new event is then added to *rwEvents* that is also cleaned up, which means that the dominated events and the old events are removed.

C. Additional Details and Remarks

Let us now discuss a number of issues that were left out of the discussion above. For example, the behavior of re-joining nodes needs to be considered. In our approach we assume that nodes that leave the network (detected as failed) will keep their *storageQueue* when joining again, but they empty their *rwEvents* cache. In addition, they move the content of their old *sendQueue* to *storageQueue*. This prevents outdated messages from arising as a result of re-joining.

It is also worth mentioning that the broadcast messages are temporary, that is, we delete them immediately after their maximal hop-count is reached. This means that without failure events no broadcasting is going on as all the caches are empty.

Let us now clarify the handling of the different restart IDs in the message processing. When determining the dominance relation and the conflicts described previously, we compare only those messages where the walk ID and the old restart ID are the same. In other words, only those messages are compared that belong to the same failure event. This means that, for example, it is possible that we have two Restarted messages with the same walk ID but with different old restart IDs, and in this case there will be no conflict. The reason is that in principle this could be a situation when the walk failed, was restarted, then failed again and was restarted again within a short time.

However, when removing conflicting walks in response to a MultipleRestarts message, we remove all the walks with the same walk ID and different restart ID irrespective of the previous (old) restart ID of the removed walk. For the above reasons, there is a tiny probability that some walks that should not be removed are in fact removed.

It is possible to handle these temporal ordering issues, but we opted for simplicity and we follow our multi-level principle, namely that problems resulting from such obscure corner cases are escalated to the next level of the system. We justify this choice in our simulation experiments.

D. Best Effort Design

We stress again that the first two levels of our algorithm may fail in various ways, many of which we have not discussed here. For example, because of the inevitable delay between a failure event and its detection, it is possible that no supervisor is present for a short interval, during which the node might fail, resulting in a failed random walk. Another example is that it is theoretically possible that no node receives the restart request that has a copy of the payload of the given walk. This could be due to the unreliability of the broadcast or to the fact that most copies were deleted from storage. It is also theoretically possible that the walk will stay alive when (mistakenly) detected as failed due to the unreliability of the failure detector, which will result in undetected redundant walks. It is also possible that a restart attempt at level two eventually fails after emitting a Restarted message (which is sent before and not after the transfer completes) and although this will be detected by the supervisor of the restarter node, this can still result in incorrectly detected multiple restarts with a very small probability. Temporary network partitioning is also a problem if the supervisor and the supervised nodes are in different partitions. The list goes on.

Obviously, one could set the goal of designing a solution with provable properties under carefully selected assumptions. In a complex problem like ours, this approach would almost certainly lead to a protocol that is very hard to implement, understand and manage.

Instead, we propose a multi-level protocol with each level doing its best and escalating any unsolved problems to the next level. The function of the levels is rather clear and all of the levels allow for failures as these will be handled by the next level. Thus, the design process of the algorithms of each level is not an "all or nothing" task but rather a multi-objective optimization process where we minimize the number of failures while maximizing the simplicity and manageability of the protocol. As for research methodology, our main tool is simulation where we demonstrate the cost and reliability of the system as a whole under realistic settings.

The design goal is to ensure that our final fallback mechanism (level three) has a minimal load. At this level, the owner of the walk (and the associated task) has to provide only minimal resources like a mobile phone for 10 minutes each day, or a very limited public cloud service. During, for example, one short daily visit by the user, the walks of the user (if any) report back to the user who can remove or restart walks as needed. Thus, the system as a whole does not require expensive infrastructure and can remain open and free for all the potential users.

Note that this approach is in line with several related studies in the area of P2P-assisted systems that use unreliable distributed protocols only as a first level and a central service provides the guarantees for the reliability of the application (for example, [15], [16]). Here our goal is slightly different in that we want to reduce the contribution of the central final level to an absolute minimum, and also in that we organize

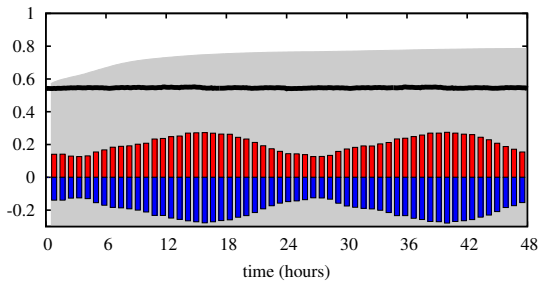

Figure 1. Proportion of users online, and proportion of users that have been online, as a function of time. The bars indicate the proportion of the simulated users that log in and log out (shown as a negative proportion), respectively, in the given period.

the distributed protocol itself into hierarchical levels in a similar manner.

IV. Experiments

In order to experimentally analyze our protocol, we simulate node churn based on a real trace of smartphone user behavior. To perform the simulations, we used PeerSim [17].

A. Trace Properties

The trace we used was collected by a locally developed openly available smartphone app called STUNner, as described previously [11]. In a nutshell, the app monitors and collects information about charging status, battery level, bandwidth, and NAT type.

We have traces of varying lengths taken from 1191 different users. We divided these traces into 2-day segments (with a one-day overlap), resulting in 40,658 segments altogether. With the help of these segments, we are able to simulate a virtual period of up to two days by assigning a different segment to each simulated node. Here, we simulated our protocol for one day, using the first half of each segment. When we needed more users than segments, we re-sampled the segments to inflate the number of users artificially.

The observed churn pattern is illustrated in Figure 1 based on all the two-day periods we identified. Although our sample contains users from all over the world, they are mostly from Europe, and some are from the USA. The indicated time is GMT, thus we did not convert times to local times. We treated those users as offline who had a bandwidth of less than 1 Mbit/s.

Note that we can simulate the case where a participating phone is required to have at least a certain battery level. From the point of view of churn, though, the worst case is when any battery levels are allowed to join, because this results in a more dynamic scenario. However, the first 10 seconds of each online session (or the entire session if it is shorter) are considered offline to remove extremely short online sessions. This technique can also be explicitly implemented as part of our protocol: a node should wait 10 seconds before joining the network.

B. Experimental Setup

All our experiments were run on top of the churn trace described above. In each experiment the algorithm parameters listed in Table I were fixed. Note that δ (the length of the restarting window) is calculated as 20Δ, which is the longest time a given broadcast message is expected to spend in the network. This increases the possibility that restarting windows are indeed sequential and non-overlapping. The maximum allowable amount of data in the storage queue is set so that the queue could store ten entries of the maximal size. Thus, in scenarios where the payload size is variable, the queue might store more than ten entries.

Table I
Fixed Parameters

Δ	100 ms
δ	2000 ms
storageQueue size	10 entries of maximal size
max gossip hop count	20 steps

In our experiments we varied three main parameters of the application environment. These were the network size, the number of random walks to maintain, and the distribution of the size of the payload of the random walks. As for network size, we experimented with $n = 1000$ and $n = 100,000$.

Regarding the number of random walks, we designed three scenarios with an exponentially increasing number of random walks. To determine how many walks to start, we first examined the churn trace and found that, on average, 54% of the nodes are online. Based on this, as a baseline setup we started $0.54n$ random walks in expectation. As for the implementation, each node was assigned a walk initially with a probability of 0.54. We also ran experiments with ten times more and ten times fewer walks than this baseline. The exact number of random walks can be found in Table II.

The payload size was defined in terms of transmission time assuming a fixed bandwidth limit at the nodes. We defined a small and a large payload with a transmission time of 1000 ms, and 10,000 ms, respectively. We ran simulations with only small and only large payloads, as well as with a mixture of payloads where each random walk was assigned a transmission time at random with a uniform distribution over the interval [1000 ms,10,000 ms].

The overlay network was implemented by independently assigning 50 randomly selected neighbors to each node. This setting was used for all the network sizes. We assume that each node maintains an active TCP connection with its neighbors as suggested in [8]. If a node fails, its neighbors will detect this only with a one second delay. The neighbor set is constant in our simulations; that is, when a neighbor fails it remains on the list and it is reconnected when it comes back online. The size of our neighbor set was large enough for the overlay network to remain connected.

We should also mention that we applied a short warm-up period before the simulation that is not included in our reported statistics. We did this to model the realistic usage scenario where new random walks are added to the system by their owners making sure that the walk completes at least a few hops so that there are copies in the network to restart from. For this reason, those nodes that were assigned a walk to start were kept online for five minutes and we started the trace-based simulation only after this period. In addition, to avoid an immediate synchronized spike in failures (a

simulation artifact), we made sure the nodes that started a walk were assigned a trace with an initial online period.

C. Results

Let us first take a look at the statistics of the various experimental scenarios at the end of the simulated day in Table II. We can see that in all the cases there is a large number of restarts, more than ten times as many as the number of walks. The most important result is that the vast majority of these restarts happen at level one.

It is clear that the number of level two restarts depends mostly on the size of the payload of the walks. With a large payload (and long transmission time) there is a larger probability that the level one supervisor fails before the transmission gets completed, thus triggering a level two process.

Also, we get a disproportionate increase in the number of level two restarts when we increase the number of walks beyond the number of online nodes. In that scenario, apart from the fact that there are more walks and thus more restarts, the walks spend a lot of time in the sending queues of the nodes, thus the expected time for the supervision becomes an order of magnitude longer. This in turn increases the probability of the failure of the level one supervisor.

As for level three events, we did not observe any instances of redundant walks, and only around one percent of the walks got lost.

A very interesting issue is the communication cost of the protocol. From the table, we can see that the maximal size of the broadcast table is extremely small; it is in fact negligible when considering that the entries in the table are also very small. Note that the maximal value is indicated, but in fact the broadcast tables are empty most of the time. This indicates that the overhead of the protocol is small, so the communication costs are dominated by the random walks.

Let us now examine the effect of the various levels. When omitting level two, not surprisingly, slightly fewer restarts happen at level one, since those walks that get lost will not need further restarts. When there is no restarting mechanism in place, we lose all the walks.

Finally, let us note that our experiment with the large network size of $n = 100,000$ also produces very few level three failures and the broadcast cost is as low as in the smaller networks. This confirms the scalability of the approach.

Table II does not illustrate the dynamic properties of the statistics, and the speed of the random walks cannot be seen either, which is a key property we wish to maximize. Thus, we include plots as well that contain the number of hops the random walks complete as well as the number of walks that are alive as a function of time.

Figures 2, 3 and 4 were obtained using the three different payload size distributions we experimented with, and each figure contains three plots that correspond to the three different numbers of random walks.

The plots illustrate how quickly the random walks die out without any restart mechanism. As we saw in Table II, here we can also observe that relying only on level one restarts (and not using level two) more walks get lost, although, as we saw previously, the difference is not large.

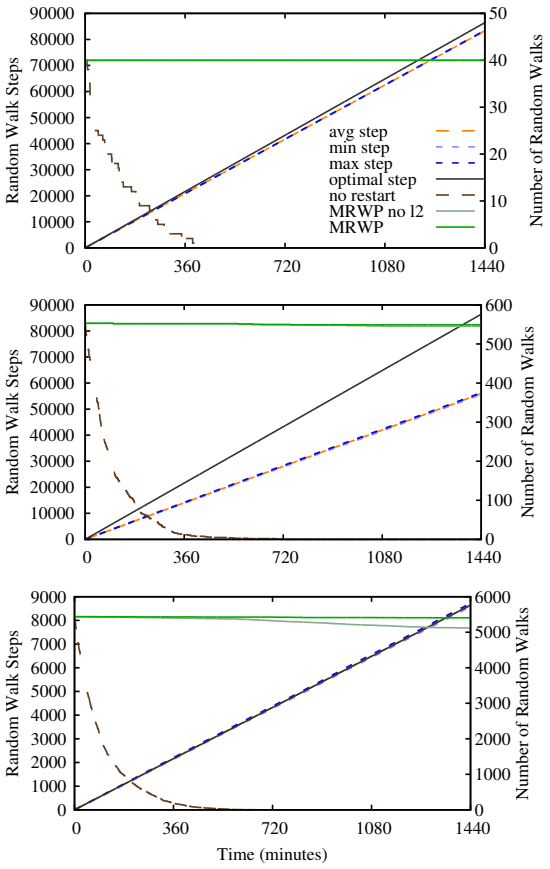

Figure 2. Experimental results with $n = 1000$, and small payload (1000 ms transmission time) with a varying number of random walks.

The plots also contain statistics about the number of hops (steps) the random walks completed by the given point in time in the form of average, minimum and maximum in the homogeneous payload scenarios and in the form of samples from individual walks in the mixed scenario. The optimal speed is also indicated. When calculating the optimal speed, we took the number of walks into account. That is, when there are ten walks on average for each online node, the optimal speed gets divided by ten to account for the fixed bandwidth limit we assume when transmitting random walks. Otherwise, the optimal speed is given by always transmitting the walk without any delay at the maximum bandwith.

Clearly, in the case of the scenarios with a small number of models we achieve nearly optimal random walk speed in the homogeneous payload size scenarios. In the case of mixed payload sizes, the speed of the walks with a large payload is close to optimal, but the walks with a small payload suffer delays due to queuing behind large payloads. Note that, although for all the walks the average queuing time for one hop is the same irrespective of the payload size, for walks with a small payload there are a larger number of hops on average so these walks spend more time in the queues in total.

In the scenario where the number of walks is the same as that of the average online nodes, walks slow down somewhat even in the homogeneous scenarios. This is because in this

Table II

Scenarios			Multiple Random Walk Protocol				without level 2 restarts		without restarts
network size	transmission time (s)	# random walks	# restarts at level 1	# restarts at level 2	lost random walks	max. # events broadcast	# restarts at level 1	lost random walks	lost random walks
10^3	1	40	672	0	0.00% (0)	0	672	0.00% (0)	100%
		553	9888	4	0.72% (4)	1	9922	1.44% (8)	100%
		5440	93620	253	0.62% (34)	3	90818	5.88% (320)	100%
	rand(1,10)	40	661	3	2.50% (1)	1	643	7.50% (3)	100%
		553	9512	20	1.98% (11)	1	9465	6.69% (37)	100%
		5440	75650	1083	0.58% (32)	7	67965	16.04% (873)	100%
	10	40	654	1	0.00% (0)	1	705	7.50% (3)	100%
		553	9011	46	1.44% (8)	1	8862	7.23% (40)	100%
		5440	79488	2055	0.75% (41)	7	65236	27.37% (1489)	100%
10^5	1	54383	923938	379	0.82% (449)	3	922568	1.43% (780)	100%
	rand(1,10)	54383	907904	2269	0.71% (391)	4	889612	4.85% (2642)	100%
	10	54383	887857	4183	0.74% (407)	5	858649	7.83% (4262)	100%

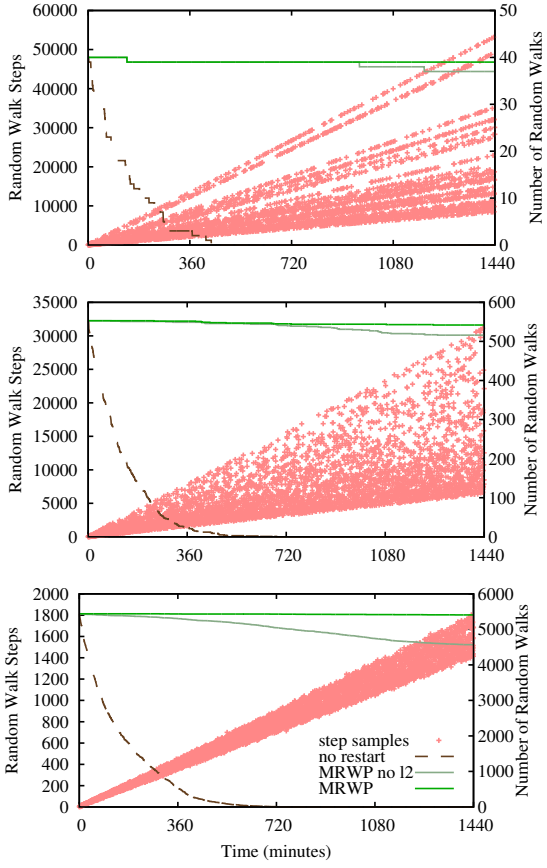

Figure 3. Experimental results with $n = 1000$, and mixed payload (between 1000 ms and 10000 ms transmission time) with a varying number of random walks.

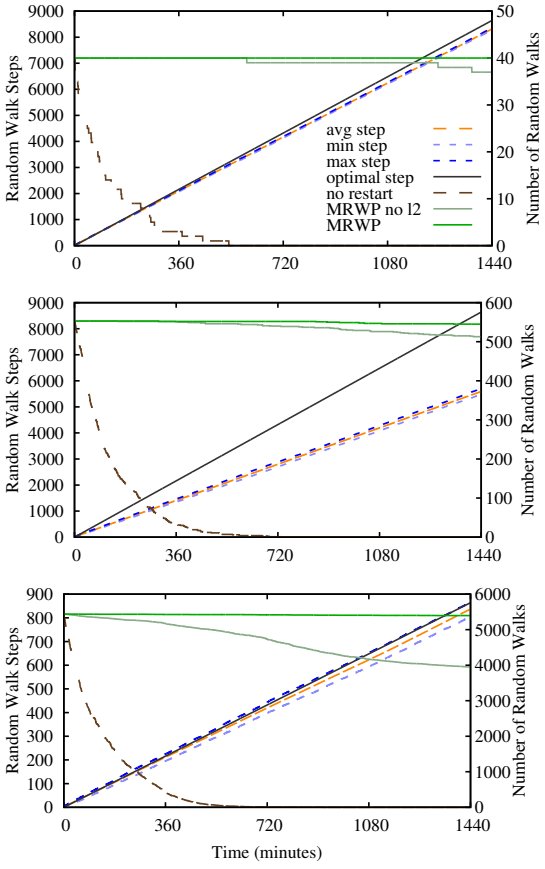

Figure 4. Experimental results with $n = 1000$, and large payload (10000 ms transmission time) with a varying number of random walks.

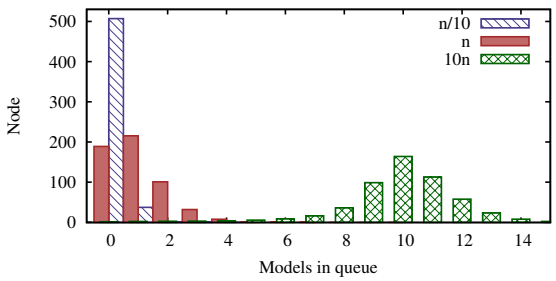

Figure 5. The histograms of the sendQueue sizes in the three scenarios with 1000 ms payload transmisson time. The notations $n/10, n$ and $10n$ represent our three settings for the number of random walks.

case the sending queue will often contain one or more walks to queue behind, which causes delays. This is illustrated well by the histograms shown in Figure 5 where the variance of the queue size can also be seen.

Interestingly, in the case of the largest number of walks the speed is close to optimal again. This is because here the queuing time is the most important factor that determines the speed and in the queues the waiting time can average out due to the larger number of walks. The histograms in Figure 5 illustrate the queue size distribution for this scenario as well.

V. Conclusions

In this study we introduced a protocol to maintain $O(n)$ random walks over an overlay network, where the random walks represent independent decentralized tasks that might belong to different users. We motivated this service with a decentralized data mining application, gossip learning, but any applications based on random walks could be supported. The protocol follows a three-level design where problems not solved at a lower level get escalated to the next level.

During our experimental evaluation we used a smartphone trace to model churn. We demonstrated that in all the scenarios we tested the vast majority of failures are dealt with at the lowest level that is purely local and therefore scalable. Only a small fraction of the problems get escalated to level two that is based on a broadcast primitive. In this case the cost of broadcast messages was shown to be almost negligible due to the small number of failure cases that reach this level. Thus, the overhead introduced by level two is small relative to the communication cost of the random walks. Level three, the central control by the task owner, was reached only a few times during all our simulations. We also demonstrated that the speed of the random walks is close to optimal.

References

[1] R. Ormándi, I. Hegedűs, and M. Jelasity, "Gossip learning with linear models on fully distributed data," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 556–571, 2013.

[2] I. Hegedűs, Á. Berta, L. Kocsis, A. A. Benczúr, and M. Jelasity, "Robust decentralized low-rank matrix decomposition," *ACM Transactions on Intelligent Systems and Technology*, vol. 7, no. 4, pp. 62:1–62:24, 2016.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. First MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. ACM, 2012, pp. 13–16.

[4] Y.-A. de Montjoye, E. Shmueli, S. S. Wang, and A. S. Pentland, "Openpds: Protecting the privacy of metadata through safeanswers," *PloS One*, vol. 9, no. 7, p. e98790, 2014.

[5] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proc. 16th ACM Intl. Conf. Supercomputing (ICS'02)*, 2002.

[6] L. Massoulié, E. L. Merrer, A.-M. Kermarrec, and A. Ganesh, "Peer counting and sampling in overlay networks: random walk methods," in *PODC '06: Proc. twenty-fifth annual ACM Symp. on Principles of distributed computing*. ACM Press, 2006, pp. 123–132.

[7] Z. Bar-Yossef, R. Friedman, and G. Kliot, "RaWMS – random walk based lightweight membership service for wireless ad hoc networks," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 5:1–5:66, 2008.

[8] R. Roverso, J. Dowling, and M. Jelasity, "Through the wormhole: Low cost, fresh peer sampling for the internet," in *Proc. 13th IEEE Intl. Conf. Peer-to-Peer Computing (P2P 2013)*. IEEE, 2013.

[9] M. Kurant, M. Gjoka, C. T. Butts, and A. Markopoulou, "Walking on a graph with a magnifying glass: stratified sampling via weighted random walks," in *Proc. ACM SIGMETRICS Joint Intl. Conf. Measurement and Modeling of Comp. Syst. (SIGMETRICS '11)*. ACM, 2011, pp. 281–292.

[10] P. Pons and M. Latapy, "Computing communities in large networks using random walks," in *Computer and Information Sciences - ISCIS 2005*, ser. Lecture Notes in Computer Science, P. Yolum, T. Güngör, F. Gürgen, and C. Özturan, Eds., vol. 3733. Springer, 2005, pp. 284–293.

[11] Á. Berta, V. Bilicki, and M. Jelasity, "Defining and understanding smartphone churn over the internet: a measurement study," in *Proc. 14th IEEE Intl. Conf. Peer-to-Peer Computing (P2P 2014)*. IEEE, 2014.

[12] A. S. Pentland, "Society's nervous system: Building effective government, energy, and public health systems," *Computer*, vol. 45, no. 1, pp. 31–38, 2012.

[13] A. Rial and G. Danezis, "Privacy-preserving smart metering," in *Proc. 10th annual ACM workshop on Privacy in the electronic society (WPES'11)*. ACM, 2011, pp. 49–60.

[14] C.-W. Tsai, C.-F. Lai, M.-C. Chiang, and L. Yang, "Data mining for internet of things: A survey," *Communications Surveys Tutorials, IEEE*, vol. 16, no. 1, pp. 77–97, 2014.

[15] A. H. Payberah, H. Kavalionak, V. Kumaresan, A. Montresor, and S. Haridi, "Clive: Cloud-assisted p2p live streaming," in *IEEE 12th Intl. Conf. Peer-to-Peer Computing (P2P)*, 2012, pp. 79–90.

[16] G. Kreitz and F. Niemela, "Spotify – large scale, low latency, P2P Music-on-Demand streaming," in *Tenth IEEE Intl. Conf. Peer-to-Peer Computing (P2P'10)*. IEEE, 2010, pp. 1–10.

[17] A. Montresor and M. Jelasity, "Peersim: A scalable P2P simulator," in *Proc. 9th IEEE Intl. Conf. Peer-to-Peer Computing (P2P 2009)*. IEEE, 2009, pp. 99–100, extended abstract.