Noname manuscript No.
(will be inserted by the editor)

PULP: an Adaptive Gossip-Based Dissemination Protocol for Multi-Source Message Streams

Pascal Felber · Anne-Marie Kermarrec · Lorenzo Leonini · Etienne Rivière · Spyros Voulgaris

the date of receipt and acceptance should be inserted later

Abstract Gossip-based protocols provide a simple, scalable, and robust way to disseminate messages in large-scale systems. In such protocols, messages are spread in an epidemic manner. Gossiping may take place between nodes using push, pull, or a combination. Push-based systems achieve reasonable latency and high resilience to failures but may impose an unnecessarily large redundancy and overhead on the system. At the other extreme, pull-based protocols impose a lower overhead on the network at the price of increased latencies. A few hybrid approaches have been proposed—typically pushing control messages and pulling data—to avoid the redundancy of high-volume content and single-source streams. Yet, to the best of our knowledge, no other system intermingles push and pull in a multiple-senders scenario, in such a way that data messages of one help in carrying control messages of the other and in adaptively adjusting its rate of operation, further reducing overall cost and improving both on delays and robustness.

In this paper, we propose an efficient generic push-pull dissemination protocol, PULP, which combines the best of both worlds. PULP exploits the efficiency of push approaches, while limiting redundant messages and therefore imposing a low overhead, as pull protocols do. PULP leverages the dissemination of multiple messages from diverse sources: by exploiting the push phase of messages to transmit information about other disseminations, PULP enables an efficient pulling of other messages, which themselves help in turn with the dissemination of pending messages. We deployed PULP on a cluster and on PlanetLab. Our results demonstrate that PULP achieves an appealing trade-off between coverage, message redundancy, and propagation delay.

Keywords Peer-to-peer network, Gossip-based dissemination, Epidemic algorithm

Pascal Felber · Lorenzo Leonini · Etienne Rivière
Université de Neuchâtel, Institut d'Informatique, Émile-Argand 11, 2009 Neuchâtel, Switzerland.
Tel.: +41 32 718 2725, Fax: 41 32 718 2701.
E-mail: {pascal.felber,lorenzo.leonini,etienne.riviere}@unine.ch

Anne-Marie Kermarrec
INRIA Rennes-Bretagne Atlantique, Campus Universitaire de Beaulieu, 35042 Rennes cedex, France.
E-mail: anne-marie.kermarrec@inria.fr

Spyros Voulgaris
Vrije Universiteit, Computer Science Department, Amsterdam, The Netherlands. E-mail: spyros@cs.vu.nl

1 Introduction

Disseminating information from and to very large communities of nodes is fundamental in many systems and for a wide spectrum of applications, such as spreading antivirus updates, propagating control messages or monitoring information (particularly when arriving in bursts), operating a content delivery network, etc.

Solutions based on *dedicated resources*, be they individual servers, server farms, or distributed architectures of dedicated forwarders, share the common issue of *cost effectiveness*, as they are extremely difficult to provision accurately. More specifically, the almost inevitable *over-provisioning of resources* raises cost dramatically. Any amount of dedicated resources has a finite limit on the load it can handle. A system claiming high scalability should provision resources for the highest possible anticipated load, even if that peak load appears very rarely, if at all. This results in a significantly underutilized system during regular operation, leading to massive waste of resources and money.

Collaborative architectures embracing peer-to-peer (P2P) models form the natural alternative to systems based on dedicated resources. Nodes making use of a service also contribute to it, relieving the total system of part of the work in a proposition comparable to the load they impose on it. This property is often referred to as *elasticity*, in the sense that system resources rapidly scale up and down along with the demand.

In this context, *epidemic* (or *gossip*) *protocols* have recently received an increasing interest. Their attractiveness stems from their scalability, inherent balancing of load across all nodes, and quick convergence. Additionally, they have proven to be remarkably robust in the face of failures. Last but not least, they are extremely simple. They rely on a periodic pairwise exchange of information between peers, and are particularly suited to implement a global emergent behavior as a result of local interactions based on limited knowledge. They have been applied to a wide variety of applications [8,13,15,17,32]. Yet, the most classical use of gossip protocols is to reliably disseminate data in large networks in a collaborative manner [3,9,16].

Gossip-based dissemination transmits information in the same way as a rumor spreads within a large group of people in real life, or a disease spreads by infecting members of a population, which can in turn infect others. Randomness and repetitive probabilistic exchanges are at the heart of gossip-based protocols and are keys to achieve robustness. Messages are relayed in an epidemic manner so that, with high probability, they are received by all peers in the system.

More recently, gossip-based approaches have been extensively used in the context of streaming applications [19,21,34]. Some recent work [4] has demonstrated that epidemic live streaming algorithms can achieve nearly unbeatable rates and delays. This growing interest in gossip-based dissemination can be explained by the more than ever dynamic nature of large-scale systems with frequent failures and unreliable communication links, emphasizing the fragile nature of tree-based approaches. These systems consider the transmission of a stream of messages from one source to a large number of consumers, typically for in-order replay. However, besides the now well-understood single-source streaming problem and associated protocols, there is also a need from applications for the support of all-to-all data transmission where messages are emitted by potentially all nodes in the system and shall be received by all others. In this context, message emissions are generally not correlated, thus a weak or no ordering is typically sufficient. Examples of such applications include wide-area monitoring, logging and update mechanisms and notification services. These applications are the ones we are targeting in this paper. They share the following common characteristics. Messages are sent by multiple sources (i.e., any peer in the system can be the source of a new message or set of messages). Messages are typically of moderate size and do not need to be sent in several pieces (or chunks), which would typically be achieved in a single-source dissemination using protocols such as BitTorrent [7]. While messages from different sources are not necessarily correlated, the overall rate at which messages are sent in the whole group is typically varying in time: burst of messages can be sent to all peers in the system at some point in time while in the common case only a few messages are disseminated per unit of time, or messages can be triggered at multiple sources in response to the reception of a previous message. Finally, the targeted infrastructure is a non-reliable one, where messages can be lost and nodes can join or leave the system at any moment. A dissemination service must take this aspect into account but at the same time achieve reliable and efficient dissemination at the lowest possible cost on the network (by minimizing the number of messages and the overall bandwidth used for the dissemination and its management).

There are two main methods for epidemic dissemination of messages, as laid out in the seminal paper on epidemics by Demers et al. [8]. The first one (referred to as *rumor mongering* in [8]) is a *reactive* method. Upon reception of a new message, a node actively pushes it forward to a few other nodes in the network, which, in turn, do the same until some termination condition is met. The second method (referred to as *anti-entropy* in [8]) is *proactive*. Each node periodically probes a random other node to check for messages it has not received yet, and pulls them if there are any. For convenience, in this paper we will be referring to these methods as *push* and *pull*, respectively.

1.1 Evaluation Metrics and Objectives

In order to evaluate the strengths and weaknesses of gossip-based dissemination protocols, we consider a set of metrics traditionally used to assess the performance of dissemination protocols, namely *delay*, *coverage*, and *redundancy*.

- **Delay** refers to the time intervening between the generation of a message and its delivery at some destination.
- **Coverage** refers to the ratio of peers that receive a message. We are targeting coverage ratios of 100%, i.e., each message should be delivered to *all* peers.

– **Redundancy** refers to the number of duplicate—and therefore unnecessary—message deliveries. Although it improves resilience to failures, too high a redundancy may overload the network.

Another important metric is that of the overall cost of the dissemination mechanism, in terms of the number of messages (for the dissemination itself, maintenance of the dissemination substrate, and control flow messages) and used bandwidth (typically dominated by redundant sends of messages). Ideally, messages originating at any peer should be delivered to *all* peers in the system (complete coverage) with reasonable delays. In our context, this should be achieved in a robust manner and with the lowest possible cost, meaning achieving a minimal redundancy and using as few messages as possible for the protocol operation.

1.2 Contributions

We start by pointing out the shortcomings of the push and the pull methods, and illustrate them by experimental results. Our contributions are then the following. We present the specificities related to the dissemination of a *flow of messages* and how forwarding at random directions can be leveraged to achieve *complete* disseminations at low cost and with low delay. Next, we present a new protocol based on our observations, Pulp, that mingles push and pull in such a way that each one makes up for the weaknesses of the other. Pulp is a highly scalable and adaptable collaborative protocol for the dissemination of multiple messages in very large sets of peers. The performance, costs, and resilience of Pulp are conveyed by real deployments under static and dynamic scenarios on a cluster and on the PlanetLab testbed [1].

1.3 Outline

The rest of the paper is organized as follows: In Section 2, we discuss the strengths and weaknesses of the push and pull approaches. The design of the Pulp protocol is presented in Section 3. In Section 4, we perform a thorough experimental evaluation of Pulp. Section 5 discusses related work. Finally, Section 6 concludes.

2 The Push-Pull Dilemma

As already mentioned, epidemic-based dissemination protocols are based on two basic methods: *push* and *pull*. In this section, we identify the strengths and weaknesses of each approach in an attempt to come up with an efficient hybrid scheme that combines the best of both worlds.

2.1 Push protocols

Push protocols are based on the recursive forwarding of messages among peers. A node receiving a message actively passes it on to a few random other nodes, which recursively do the same until some termination condition is met. The termination condition ensures that the recursion does not go on forever. For instance, messages could be augmented by a Time-to-Live (TTL) field to limit the number of hops they can take. Alternatively, nodes could be programmed to forward messages only upon their first reception and ignore subsequent copies. Either solution ensures that the dissemination of a message eventually fades out. The number of times an informed node forwards a message is denoted as the Fanout.

Regardless of the specific variation of the push protocol, reaching *all* nodes by blindly forwarding a message in random directions is a very expensive operation. Assume a rather ideal and generic model for push dissemination (we explain later why reality is harsher), where nodes are selected *uniformly at random* and *one at a time*, out of a total population of n nodes. At each iteration, the message is forwarded to the selected node, independently of whether it is already informed.

The probability to select a not-yet-informed node when k nodes have already been informed is $\frac{n-k}{n}$, which requires an expected number of $\frac{n}{n-k}$ random forwards to reach the $(k+1)$-th node. This number lies between one and two until half of the nodes have been informed, but increases dramatically for the last few nodes. For instance, reaching the *last* node alone requires on average n forwards. The expected number of times a message should be forwarded to reach the whole population of n nodes is

$$\sum_{k=0}^{n-1} \frac{n}{n-k} = n \cdot \sum_{i=1}^{n} \frac{1}{i} \approx n \ln n + \gamma n$$

for high values of n, where $\gamma \approx 0.5772$. That is, the expected total number of forwards to reach all nodes is in the order of $\mathcal{O}(n \ln n)$.[1]

Reality, however, is harsher. First, forwarding a message $n \ln n$ times *does not ensure* that it will reach all

[1] These probabilities are studied in the equivalent *Coupons Collector* problem, where a collector keeps selecting at random out of n different coupons with replacement, and the number of trials until all coupons have been selected at least once is measured.

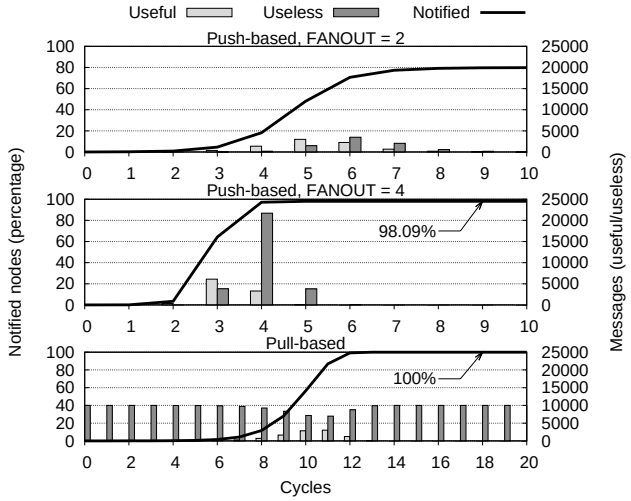

Fig. 1 Discrete time simulation of redundant ("useless") message delivery ratios and coverage for push-based and pull-based epidemic diffusions in a 10,000 node network. TTL is ∞ for the push-based simulation.

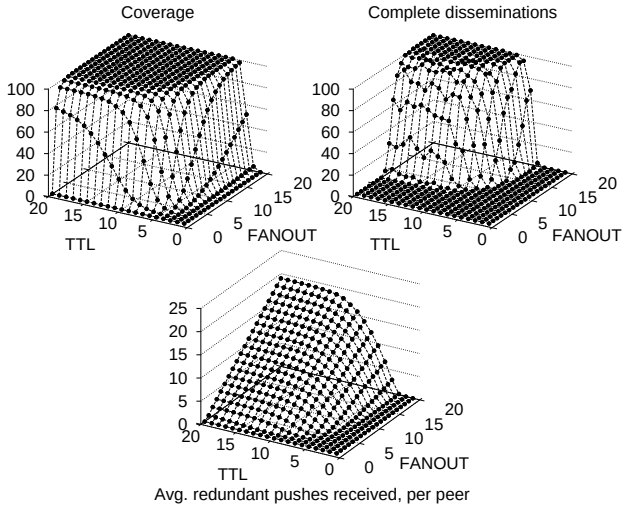

Fig. 2 Push-only: Coverage, number of complete disseminations and average number of redundant (useless) messages received per peer, as a function of the TTL and FANOUT.

nodes. Second, as dissemination is carried out in a totally decentralized, asynchronous, and massively parallel way, it is not possible to impose fine-grained control on the number of times a message is forwarded. As a result of these, the indicated value $n \ln n$ often ends up being significantly surpassed, resulting in significant additional redundancy.

2.2 Pull protocols

In a pull protocol, each node periodically probes random peers in the network in hope to reach an already informed peer, and retrieves new messages when available. Typically, during a pull round, random pairs of peers exchange information about the messages they have recently received and request missing messages from each other.

Contrary to push protocols, the probability of an uninformed node receiving a message increases linearly with the current coverage of the message. Indeed, if k out of n nodes are informed, then a non-informed node will probe an informed one with probability $\frac{k}{n-1}$. In the case of the *last* uninformed node, it will pull the message with probability 1 the next time it probes a random node, as all other nodes already have it.

2.3 Coverage versus redundancy

In order to experimentally validate the aforementioned properties, we consider the following push protocol. Each message is augmented by a TTL value, determining the number of hops it can traverse. A node receiving a message for the *first time* decreases its TTL by one, and if it is not lower than zero forwards the message to FANOUT random other nodes. When a node receives a message that it has already received (and possibly forwarded) in the past, it simply ignores it.

Figure 1 presents a simulated overview of the behavior of push- and pull-only protocols, with respect to coverage and redundancy. For the sake of simplicity, both protocols are presented in a synchronous way: all peers that pull from, or push to, another node do so at the same time. This synchronous activity is called a *cycle*. We consider FANOUT values of 2 (top) and 4 (middle), with infinite TTL in push protocols. A random node sends a message at cycle 0.

We observe that in push protocols the data spreads exponentially fast through the network, especially in the first rounds. However, as rounds advance, the rate of the dissemination diminishes and the cost of reaching additional nodes increases drastically (as shown by the number of redundant messages, i.e., messages pushed to already informed nodes). A higher FANOUT produces a sharper exponential growth of the set of reached nodes, but also a higher level of redundancy. One can notice that messages are not pushed to all nodes.

Regarding pull protocols, Figure 1 (bottom) shows that it takes several rounds until the dissemination of a message starts taking off, but once it has reached a sufficient number of nodes it quickly spreads to *all* the remaining ones. One can also observe that a constant number of control messages are sent during each round,

with the ratio of useful messages growing only during the peak of the message spread.

Figure 2 sheds more light on the behavior of push protocols, with each dot representing a separate run of a push-only dissemination for a certain combination of TTL and FANOUT values. Even if a coverage of close to (but less than) 100% can be achieved with a relatively low TTL and FANOUT, as illustrated in Figure 2 (top left), the probability of a message reaching *all* nodes is practically zero, unless both parameters have substantially higher values (top right). Unfortunately, the values for TTL and FANOUT that provide good coverage properties produce high redundancy, expressed as the number of deliveries per message per node (bottom).

2.4 Delay

Push protocols deliver messages with a relatively low delay, as they can immediately push messages further upon reception at each step of the dissemination, in an avalanche-like manner. Latency is therefore directly proportional to the network delays and the number of hops from the source. In contrast, pull protocols can exhibit high latency because, even if the propagation time of messages does also depend on the number of hops from the source, it is multiplied by the pull period. A high frequency will produce much (unnecessary) traffic while a low frequency will dramatically increase latency.

2.5 Discussion

While both push and pull protocols may achieve full coverage, they do so through complementary patterns. Push quickly spreads messages to a large portion of the network, as it does not depend on timing assumptions (e.g., no periodic operation). It is, however, slow and prohibitively expensive in reaching the last few nodes. This renders it an excellent candidate for the early stages of dissemination, but inappropriate for the final phase.

Pull, on the contrary, is an excellent candidate for the final stages of dissemination, as it deterministically delivers each message to *all* remaining nodes in logarithmic steps, and by pulling messages selectively it eliminates the problem of redundant message forwarding. However, it is a poor choice for the early stages, as it starts very slowly. The main disadvantage of pull, though, is that probing requests are periodic, generating a non-negligible *steady state load* proportional to the probing frequency. However, lowering the probing frequency to save on traffic overhead increases the dissemination delay, leading to delicate tradeoffs.

These complementary patterns of push and pull are the driving force behind the PULP protocol, described in the following section.

3 The PULP Protocol

From the observations presented above, it appears clearly that there is no ideal protocol performing well on all fronts. In this section we present PULP, a hybrid protocol harnessing the specific strengths of both approaches.

3.1 System Model

We consider a large set of n nodes communicating over an unreliable, fully connected medium (e.g., UDP over the Internet). Nodes can join or leave the network at any time. Departures and crash failures are treated equivalently, that is, there is no *graceful leave* operation. Byzantine behavior is out of the scope of this paper (see, for instance, BAR Gossip [21] for a dissemination protocol dealing with byzantine nodes).

All operations are *fully decentralized*. That is, there is no central entity to control any function of the system. All message exchanges (both periodic and sporadic) between nodes are *asynchronous*. Note that the dynamic and unreliable nature of the network rules out protocols that depend on rigid structures or reliable communication channels, such as tree-based dissemination protocols using TCP communication.

Regarding the anticipated workload, we consider (1) a *sequence of messages* being disseminated rather than a single message, (2) generated at variable arbitrary rates, and (3) originating at multiple nodes. As we will see, point (1) is particularly important as message disseminations are leveraged to inform nodes of previous messages that might have been missed, which in turn helps nodes adjust their pulling frequency. The resulting protocol is adaptive and self-controlled based on the current message generation rate. Note that the model of a sequence of messages matches the nature of many common applications, such as microblogging, RSS feeds, etc.

3.2 Supporting Mechanisms

Like many epidemic protocols, PULP relies on communication between peers selected *uniformly at random*. To that end, we rely on the family of PEER SAMPLING SERVICE protocols [13], and specifically CYCLON [33],

which provides each node with a regularly refreshed list of links to random other peers, in a fully decentralized manner and at negligible bandwidth cost.

To provide a high level sketch of CYCLON we omit certain details found in [33]. In a CYCLON overlay, each node maintains a (very short) partial *view* of the network, that is, a handful of links (IP addresses and ports) to other nodes. Periodically, yet asynchronously, each node contacts a peer from its view, and they exchange a few of their views' links. As a result, views are periodically refreshed with new links to random other peers of the overlay. When the right policies are followed (see [13] and [33] for details), this method has shown to produce overlays that strongly resemble random graphs, that is, at any given moment each node's view contains links to nodes selected uniformly at random from the whole network. Moreover, this process has shown to converge in a few dozen cycles irrespectively of the initial topology, and due to the self-healing nature of CYCLON the respective properties are retained even in the face of node churn[2]. CYCLON and most other PEER SAMPLING SERVICE protocols have negligible computational, memory, and bandwidth cost,[3] and have shown to operate with remarkable reliability and robustness in (even highly) dynamic conditions.

To elaborate on the feasibility of nodes to communicate with randomly selected peers, when a node is equipped (through CYCLON) with a few links to *randomly selected* other nodes, and it *randomly selects* one among them, it is equivalent to having selected one node *at random* from the whole overlay. Further, when the node's CYCLON view is changing over time, the node has essentially access to an endless stream of random peers to communicate with.

Note that peer sampling protocols are also able to cope with the characteristics of actual IP networks, in particular with respect to nodes' reachability (as the node lies behind a firewall or NAT). The authors of [**?**] propose an augmentation of the CYCLON protocol that also deals with NAT-traversal issues while maintaining the same randomness characteristics for the overlay.

Finally, nodes in PULP need to have a *rough* estimate of the network size. For that, we employ the *interval density algorithm* [18], which can be executed *locally* on top of CYCLON with negligible cost. The principle of this algorithm is based on the fact that the CYCLON view provides a continuous stream of randomly selected peers from the entire network. Estimation of the network size relies on the density of these peers over a chosen value space, and proceeds as follows. Each node applies a hashing function to the IP addresses of each peer it discovers through CYCLON, mapping them to values uniformly spread in a given value space. It keeps a set of recent peers' hashed values that are the closest to a particular value (e.g., its own hash value), and uses the span of this set over the whole value space to infer an indication of the network size. More than one hashing function can be used for increased accuracy.

[2] In our experiments CYCLON converged in no more than 20 "cyclon rounds", that is 100 sec, and remained converged thereafter even at experiments involving churn.

[3] In our experiments CYCLON traffic accounted for an average of 24 bytes/sec per node, as explained in Section 4.1.

3.3 PULP: The Intuition

As explained extensively in Section 2, push-based and pull-based protocols operate with opposite patterns. Conveniently enough, these patterns are complementary with each other regarding their strengths and weaknesses.

Given the respective observations, PULP strives to meet the following objectives:

Limit push. Let push execute only for the very few initial steps, to avoid redundant message forwarding while ensuring sufficient startup diffusion of messages. Our experiments (Section 4) indicate that reaching 4% to 5% of the network is a good target for the push phase, with nearly no redundancy.

Reduce redundant pulls. Avoid probing to find out *whether* a message is missing. Probe only in an attempt to pull the message, when it is known to be missing.

Adapt the pull period. Periodic probing constitutes a limitation. Too short a period causes unnecessary probing message load, particularly in periods of low message rate. Too high a period renders the system unresponsive when messages come at high rate. PULP is designed to dynamically adjust the probing frequency of nodes to match the current message rate.

The key observation is that if messages are forwarded to nodes selected uniformly at random, every message reaches a different set of nodes that is not correlated to the sets of nodes reached by other messages. Although a node might miss a given message with significantly high probability, the probability of it missing *all* of k messages, diminishes exponentially with k. We exploit this property in our algorithm.

With respect to the first objective, reaching 4% to 5% of the network in the initial push phase requires to

set the values of TTL and FANOUT accordingly.[4] The coverage obtained depends on both of these parameters, as well as the size of the network, which is estimated by the interval density algorithm (see Section 3.2). A node with a size estimation N_{est} simply chooses, for messages it generates, the values of TTL and FANOUT such that

$$c_{\text{est}} = N_{\text{est}} / \sum_{i=1}^{TTL} Fanout^i$$

is as close as possible to the expected coverage. Either TTL or FANOUT is fixed (possibly based on allowed range of values or on the CYCLON view size for the FANOUT) and the other parameter is derived accordingly. Our current implementation fixes the FANOUT to 3 and computes the value of TTL. Initial push messages can be sent with different TTL values to approach more closely the required coverage. Note that duplicates can be ignored in the calculation, as the value of the expected coverage is indeed required to be low enough to actually *avoid* most duplicates.

Regarding the second and third objectives, PULP leverages the push phase to relieve the pull phase of excessive probing requests. Instead of having each individual node explicitly probe random other nodes at fixed intervals to discover *whether* it is missing any messages, forwarded messages carry information about which other messages are available, conveying this information as a by-product of the push component.

3.4 PULP: The Protocol

We now present a detailed description of the PULP algorithm, which combines the push and pull components for disseminating a sequence of messages in a collaborative and decentralized fashion.

Algorithm 1 shows the pseudo-code of the PULP protocol. Each peer P maintains a *history* of the messages it has recently received, denoted as H_P. It additionally maintains a *trading window*, denoted as T_P, containing the list of messages that are available to other nodes on request.

When a message is pushed to (or generated at) node P for the first time, P registers it in H_P and, if the TTL has not been reached yet, forwards it to FANOUT random other peers. We stress that obtaining the IP address of randomly selected peers is a trivial task thanks to CYCLON, as described in Section 3.2.

[4] We chose this value of 4% to 5% as they allow for a low latency dissemination with only very few duplicates. Using larger values do not reduce the delays further but significantly increase duplicate counts. This choice is experimentally justified in Section 4.2.

Algorithm 1: Pulp algorithm on node P

```
Variables
    H_P: History of (recently) received message IDs
    Δ_pull: Period of pull operations (initially 30s)
    missing: Set of message IDs known, but not yet received
    prevMissingSize: Size of missing at the end of last Δ_adjust period
    prev_useful: Number of useful pull replies during current Δ_adjust period
    prev_useless: Number of useless pull replies during current Δ_adjust period
(Δ_adjust, TTL and FANOUT are fixed protocol parameters)

// Invoked when a message is pushed to node P by node Q
function PUSH(msg, hops, Q, T_Q)
    // Forward further if needed
    if msg received for the first time then
        add msg to H_P
        if hops > 0 then
            invoke PUSH(msg, hops-1, P, T_P) on FANOUT random peers
    // Messages will be pulled at the next pulling period
    missing ← missing ∪ {m ∈ T_Q : m ∉ H_P} \ {msg}

// Periodic pulling of missing elements
thread PERIODICPULL()
    do every Δ_pull seconds
        // Shuffling reduces the probability of receiving duplicates by pull
        shuffle missing
        invoke PULL(missing, P, T_P) on a random node Q

// Invoked when a node Q requests a message from node P
function PULL(requested, Q, T_Q)
    m ← 1st element in requested order ∈ T_P, or ⊥ if none
    invoke PULLREPLY(m, P, T_P) on Q

// Receive a reply to a pull request from node P
function PULLREPLY(msg, Q, T_Q)
    if msg = ⊥ ∨ m ∈ H_P then
        prev_useless ← prev_useless + 1
    else
        add msg to H_P
        missing ← missing ∪ {m ∈ T_Q : m ∉ H_P} \ {msg}
        prev_useful ← prev_useful + 1

// Periodic adjustment of pulling period for node P
thread ADAPTFREQ()
    do every Δ_adjust seconds
        if |missing| > prevMissingSize then
            Δ_pull ← Δ_adjust / (|missing| − prevMissingSize + prev_useful)
        else
            if |missing| > 0 ∧ prev_useless ≤ prev_useful then
                Δ_pull ← Δ_pull × 0.9
            else
                Δ_pull ← Δ_pull × 1.1
        Δ_pull ← max(Δ_pull, Δ_pull_min)
        Δ_pull ← min(Δ_pull, Δ_pull_max)
        prev_useless ← 0
        prev_useful ← 0
        prevMissingSize ← |missing|
```

In forwarding a message to another peer Q, node P also forwards the IDs of messages in its trading window T_P. These are messages that P considers to be in the pull phase, a subset of messages in P's history H_P. The trading window plays a key role in the interaction between nodes, because it helps nodes avoid exchanging messages that are either (1) too recent and are still being pushed, or (2) too old and have already been

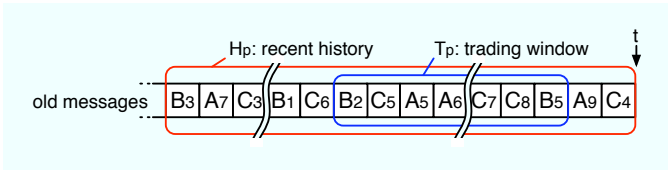

Fig. 3 Data structures of the PULP algorithm. Note that messages come from multiple sources (here A, B, and C) and each node sorts them based on the order it received them (which is generally different for each node).

removed from local histories. The trading window T_P essentially leaves a "safety margin" on both sides of history H_P. Figure 3 illustrates these data structures.

When Q receives T_P, it checks for messages not contained in its own history H_Q. If it discovers some messages it has missed, it inserts them in the *missing* set. These messages will be asked for by the periodical pull thread.

The periodic pull thread simply selects a random peer and sends it a pull request. The protocol does not try to pull from peers that are known to have the requested messages, neither does it keep information on which peer advertises what content. The rationale behind this design choice is that, while pulling from specific peers might slightly speed up dissemination of the first few messages, random selection has the advantage of distributing information about message availability more evenly (advertisements are sent along with pull requests), overall making the protocol more responsive and globally more efficient. It is a design decision that favors common benefit over (short term) individual gain.

As multiple pull operations of the same node may overlap in time, it makes sense to avoid requesting the same message from two different peers. Therefore, the set *missing* is rotated by one position (round-robin) before being sent next time. The inquired node selects, among the messages in its history, the *first* available one according to the ranking in the received *missing* list, if such an element exists. If none of the messages in *missing* is available, it replies with $\perp$, and the pull operation counts as useless. Note that, as for pushed messages, a node replying to a pull request also piggybacks its trading window in the answer.

The periodic adaptation of the pulling frequency is performed by a separate thread in the following way. Each node has a pulling frequency Δ_{pull}, and maintains a set of message identifiers, *missing*, that it has heard of but not received yet. Pulling frequencies are chosen to follow the overall rate of new messages sent in the network. Every Δ_{adjust} seconds, a node inspects (1) the evolution of the size of the *missing* set during the last period, and (2) the number of useful and useless pulls that were performed during that period. If the size of the *missing* set has increased, Δ_{pull} is lowered to the period that would have been necessary to retrieve all newly known elements (assuming successful pull operations). That is, the new pulling frequency Δ_{pull} is simply the adjustment frequency Δ_{adjust} divided by the number of messages that should have been fetched during the last adjustment period to cope with a steady rate of reception (which would result in a steady size of the *missing* set).

This adaptation allows us to cope with increasing rates of new messages and to limit the growth of the *missing* set. If the size of *missing* has shrunk, the evolution depends on the ratio of useless vs. useful pull operations: if useless pulls dominate, Δ_{pull} is decreased by a small factor. If useful pulls dominate, Δ_{pull} is increased by the same factor. We observed based on preliminary evaluations that a value of 10% was yielding a good compromise between the reactivity and the accuracy of the pulling frequency adaptation mechanism. In other words, the pulling frequency aggressively adapts to sudden sending activity and adjusts to the highest possible value with acceptable useless pulling rate. Δ_{pull} can be bounded by $[\Delta_{\text{pull}_{\text{min}}}, \Delta_{\text{pull}_{\text{max}}}]$, depending on the underlying network properties and the desired reactivity of the system to new message sending activity.

4 Evaluation

This section describes experimental results from real deployments of PULP on PlanetLab [1], as well as a controlled deployment in a cluster. We first present our experimental setup and evaluation metrics. Then, we present experimental results demonstrating the performance, stability, and load in both static and dynamic environments. Finally, we compare PULP to a push-only and a pull-only protocol to highlight the benefit of its hybrid approach.

4.1 Experimental Setup

Our implementation uses UDP for communication. UDP, being a connectionless protocol, seems the most appropriate choice given the many short communication sessions between arbitrary nodes, rather than long sessions between fixed pairs. In addition, given the inherent fault tolerance of PULP, there is no reason to opt for a more reliable (and expensive) protocol such as TCP. The ability of gossip-based dissemination to gracefully deal with packet loss is demonstrated in our experiments.

In all experiments, messages are 8KB in size and are sent from randomly chosen nodes of the network. Unless specified otherwise, the initial push phase is configured to reach between 4% and 5% of the network when there is no message loss (thus slightly less in practice). To that end, we set the parameters as follows: TTL=3 and FANOUT=3 in a 1,000 node network (ideally reaching 40 nodes—coverage 4%) and TTL=3 and FANOUT=2 in a 300 node network (ideally reaching 15 nodes—coverage 5%). We further justify this choice of a very low coverage of the initial push phase with our second experiment. Unless otherwise noted, the minimal and maximal pull periods are set to $\Delta_{\text{pull}_{\text{min}}} = 0.2$ seconds and $\Delta_{\text{pull}_{\text{max}}} = 30$ seconds.

No node has global knowledge of the network, and the selection of random peers for PULP operations is based exclusively on CYCLON [33], as explained in Section 3.2. CYCLON performs periodic pairwise shuffles of peers' views to maintain a constantly evolving overlay network whose properties are close to those of a random graph (i.e., each node's link is equally likely to be present in any other node's view). Views were configured to a size of 25 links to other nodes, and peers exchanged 5 links every 5 seconds. Given that a link is 6 bytes long (IP address and port), this accounts to 60 bytes of traffic (inbound and outbound) induced by each node every 5 seconds. Since this traffic affects two nodes, each node is involved on average in 120 bytes per 5 seconds, that is 24 bytes per second of total traffic for each node. Also, it should be noted that thanks to CYCLON's link aging policies, which are out of the scope of this paper, links to failed nodes can remain in other peers' views no more than 5 exchange rounds, that is, 25 seconds in our experimental settings.

We evaluate PULP along the metrics laid out in the introduction, namely coverage, dissemination delays, and redundancy (both in terms of redundant pushes and useless pulls). We evaluate (1) delays and their distribution, (2) the influence of the initial push phase, (3) the influence of high levels of churn on update reception delays and (4) the effectiveness of the self-adaptation of pulling periods for varying message generation rates.

4.2 Homogeneous Settings and Churn Resilience

Our first set of experiments was conducted on a local cluster composed of 11 dual-core nodes with 2 GB of memory each. Each machine hosts 91 instances of PULP, reaching a total of 1,001 nodes.

Our first experiment (Figure 4) presents the delivery delays for a stream of 200 messages. Messages are sent by randomly chosen peers at a rate of one every

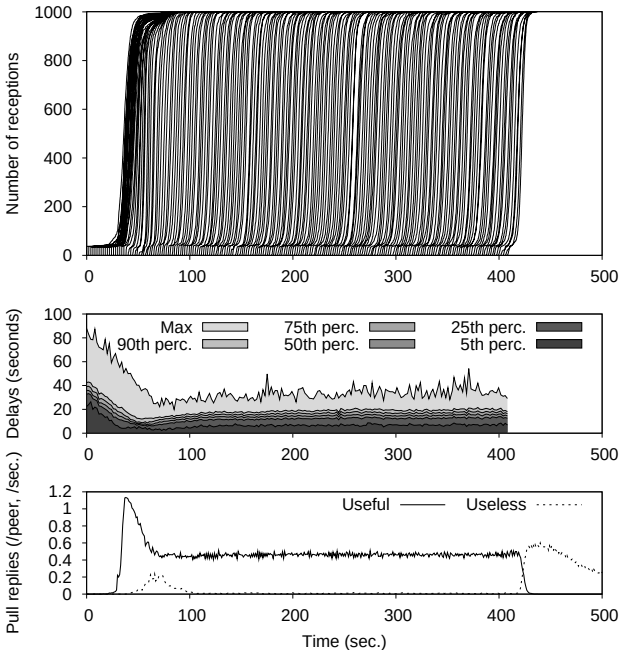

Fig. 4 Performance of the dissemination of 200 messages on a network of 1,001 nodes running on a cluster: individual cumulative delays, evaluation of the delay distribution, and evolution of the pull operations recall.

2 seconds approximately.[5] The upper plot presents the number of nodes informed (through either push or pull) for each message with respect to time. Each single line corresponds to the evolution of a certain message. The middle plot presents the distribution of delays for each message. The abscissa corresponds to the time when the message is sent. For a given abscissa, the cumulative shaded areas represent the distribution of delays, by percentiles. For instance, the maximum delay for receiving the message sent at time 200 is 31 seconds, and half of the peers receive it within 14 seconds (50th percentile). One vertical set of percentiles (distribution) in the middle plot is a concise representation of the cumulative distribution of reception times for this message, shown by one individual line in the upper plot. Finally, the lower plot presents, for each period of one second, the mean number of useful and useless pull operations performed, per peer. This metric is used by nodes to self-tune the pulling algorithm, with the objective to reach a larger number of useful than useless pull operations.

We clearly see in Figure 4 (top) the initial push phase that reaches a small portion of the network (4%): the small vertical lines at the beginning of each diffu-

[5] Due to the high load on our cluster, periods are not exactly respected by the machine's scheduler. This explains that the last message is sent at around 408 seconds and not 400 as expected.

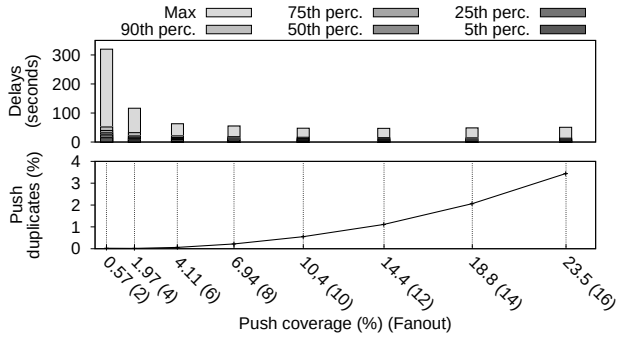

Fig. 5 Evolution of the reception latency distribution w.r.t. the coverage of the initial push phase.

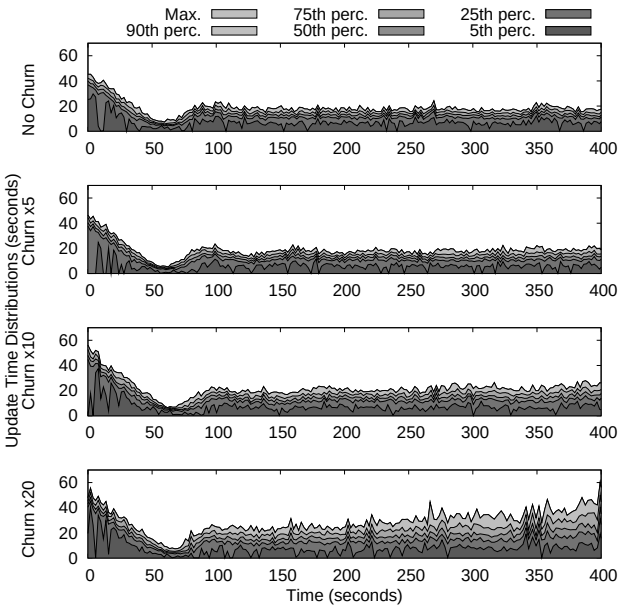

Fig. 6 Evolution of the delays as seen by a set of 100 *observers* (static nodes) under increasing churn rates.

sion, that is shown for all messages. The larger part of the dissemination takes place by pull operations. We observe an initial *bootstrap* phase, where mostly only push operations reach nodes, and no pulls take place. This is due to the fact that, prior to the dissemination of the first message, all nodes have pulling periods of $\Delta_{\text{pull}_{\max}}$ =30 seconds in this experiment. As soon as enough nodes have been reached by initial pushes, there is a *warm up* phase with an increasing rate of periodic pull operations: nodes start to discover missed messages and to retrieve them. This is also demonstrated by Figure 4 (bottom): after the warm up phase, nodes start issuing pull operations, most of which are useful (pulling too early, on the opposite, would have resulted in a larger set of useless communications). As a small number of useless pull operations appear, the pull period gracefully adapts to the message emission frequency and only a very small fraction of pull operations are useless. As a result, the delivery delays for messages sent after the warm up phase remain stable.

The next experiment explores the impact of the initial push phase on the overall delivery latency: does a larger coverage during the push phase result in lower delays? Figure 5 (top) presents the distribution of delays for a set of 200 messages in the same settings as for the previous experiment. We vary the coverage by setting TTL=2 and by varying FANOUT from 2 to 16. The lower part of the figure presents the evolution of the proportion of duplicate messages as a function of the coverage of the initial push. For instance, we have a coverage of 14.4% of the nodes on average with FANOUT=12 and this results in 1.11% of the nodes receiving at least one duplicate.

We observe the typical exponential growth of duplicates with respect to the coverage of the push phase: the higher the FANOUT, the more peers are reached, but redundancy grows faster than coverage. Most importantly, the delays that are observed with increasing coverage are decreasing only at very low values of FANOUT, and the price in redundant push operations overwhelms the benefits of higher values. This justifies the value of approximately 4% chosen as target coverage for our experiments.

4.3 Performance under Churn

The next experiment studies the resilience of the PULP protocol to churn (Figure 6). To that end, we use a *churn manager* that can emulate node departures and arrivals in real time, by remotely starting and killing our prototype nodes at specific times. We replay a real trace of 2,000 nodes collected in the Overnet file-sharing network in 2004 [2] at its original speed, as well as 5, 10, and 20 times faster. The most accelerated run result in churn rates of 192 departures or joins per minute on average for an average population of 650 simultaneously active peers.

In addition to the nodes of the trace, we use a set of 100 static nodes, denoted as *observers*, to monitor the dissemination and reception of messages. A set of 200 messages is sent by nodes belonging to the non-observer set, one every 2 seconds. We plot the evolution of delays at the observers, for increasing churn rates, and for a static environment (with 650 nodes) as a baseline. As already mentioned, communication is carried over UDP, without ACKs being sent. Failed nodes are gradually removed by CYCLON only. This means that most nodes

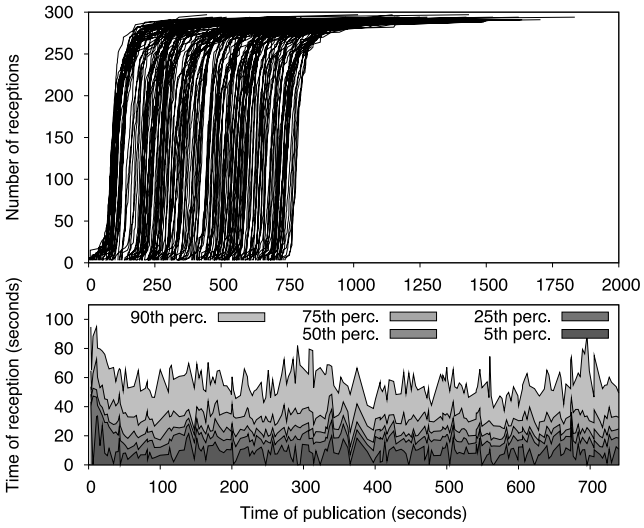

Fig. 7 Performance of the dissemination of 200 messages on PlanetLab.

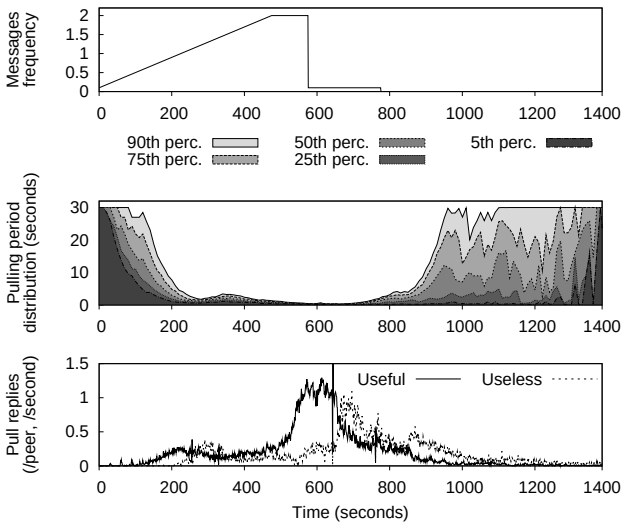

Fig. 8 Evolution of the pulling frequency w.r.t. new messages frequency.

will have (temporarily) failed peers in their views, which results in additional message losses.

Figure 6 presents the evolution of the delays and illustrates the inherent capacity of PULP, to cope with dynamic environments. We observe that only a very high churn rate leads to slightly increased delays (lower plot of Figure 6). Interestingly, nodes pull more often when there is more churn because the period Δ_{pull} decreases as more messages are lost but the size of the *missing* set does not diminish. This demonstrates that the self-adaptation of the pulling period Δ_{pull} also deals with increasing loss in the communication and keeps the rate of reception sufficiently high for all online nodes to receive all published messages.

4.4 PlanetLab Experiments

Our second set of experiments is run on the PlanetLab world-scale distributed testbed. PlanetLab is composed of nodes that are extremely heterogeneous in load and available network resources. We use a set of 300 randomly chosen PlanetLab nodes. We evaluate dissemination delays and analyze the self-tuning of the pulling period with varying message emission frequencies.

The experiments carried out on the Planetlab testbed highlight the ability of PULP to achieve full coverage in heterogeneous environments that are prone to failures, message loss, and arbitrary delays. As a matter of fact, Planetlab nodes experience significantly less reliable IP communication than typical computers on the Internet, due to their massively parallel virtualization and high load.

The first experiment reproduces the scenario studied in the cluster and shown in Figure 4, with the notable difference that we use here 300 *distinct PlanetLab nodes*. The data representation is the same as for Figure 4. A set of 200 messages are sent by nodes selected randomly at a rate of one message every 3 seconds (again, the time span deviation from 600 to 730 seconds is due to scheduling issues on heavily loaded nodes).

One can observe in Figure 7 that, as before, the first messages help to *bootstrap* the dissemination process by notifying nodes of some publishing activity. Messages then need approximately 30 seconds to reach half of the network and all nodes but the 10% slowest ones receive them in less than 60 seconds on average. Note that some of the randomly selected PlanetLab nodes failed or became unresponsive during our experiments, as one can observe in the figure: the protocol achieves a coverage of 100% of all live nodes at slightly less than 300 receptions.

Our second PlanetLab experiment evaluates the capacity of PULP to self-tune the pull period Δ_{pull} on each peer. We consider again a network of 300 PlanetLab nodes with only one publisher, whose message sending rate follows the frequency evolution of the upper plot of Figure 8: starting from a frequency of 0.2 (i.e., one message every 5 seconds) and increasing to 2 messages per second, for a duration of 475 seconds (100 messages). This steady increase is followed by the sending of 200 additional messages at a rate of 2 messages per second, followed by a sudden drop to one message every 5 seconds for the last 20 messages. 320 messages are sent in total.

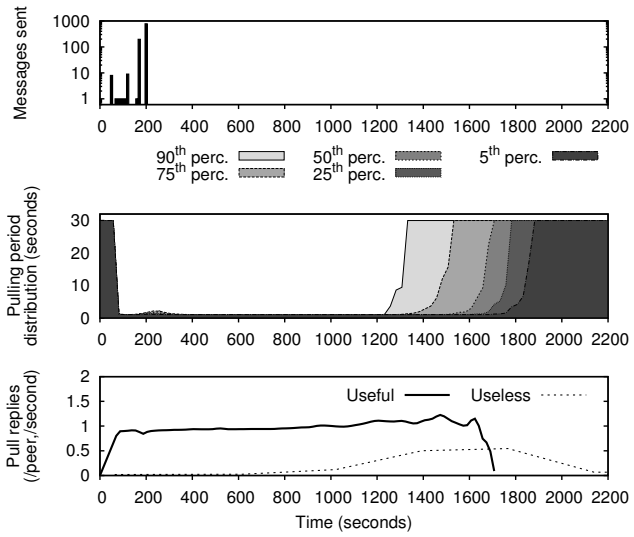

Fig. 9 Reaction to a message burst with a one second minimal pulling frequency.

We observe in the middle plot the distribution of the pulling periods Δ_{pull} computed by the self-adaptation algorithm: the initial frequency is of one pull every 30 seconds (*idle* state). As soon as messages are published, nodes discover that they miss some newly published messages and adapt their frequencies accordingly. The lower plot presents the number of pull requests issued by nodes, distinguished between useful (i.e., resulting in the reception of new information) and useless requests (i.e., retrieving no new message). We observe that the pulling frequency follows the evolution of the new messages frequency in the first phase, with a slightly oscillating behavior around the optimal value. This oscillation is typical of the control-loop-based adaptation of the pulling frequency implemented by Pulp, and is also a result of the latency between the observation and the adaptation (as adaptation decision are made periodically, on average half such a period is required before the adaptation takes place). This does not result in significant delay increases for individual messages. Then, as messages are being disseminated, most pull operations are useful and the pulling frequency remains high. As soon as most of the messages have reached all nodes, the number of useless pulls grows, resulting in an increase of the pulling period.

Overall, we observe that most of the pulls are successful and result in delays low enough to sustain the message sending rate during dissemination, whereas a pull-based protocol with a fixed pulling period would have either incurred high delays or a high number of useless pulls. This observation further highlights the importance of the self-adaptive pulling period.

Our final experiment evaluates the capacity of Pulp to react to bursty message generation scenarios. It complements the previous experiment (Figure 8). Figure 9 presents the dissemination of a total of approximately 2,000 messages in a network of 200 nodes. To better observe the behavior of the dissemination after the burst, we voluntarily slow down the system reaction by setting a very conservative minimum period for pull replies of one second. The maximal period is maintained at 30 seconds. Note that the dissemination speed in fully-loaded mode (i.e., when many messages are on-the-fly in the network and are still being propagated) is a direct function of this parameter, which in turn proportionally impacts the dissemination time.

Messages are sent from all nodes in the following manner: an initial message is sent by a first node, and other nodes react to the first message reception by triggering between 1 to 20 messages (their number is chosen randomly). This scenario illustrates the behavior of Pulp when messages are sent as bursts by combinatorial reaction. We observe on the upper plot the number of messages sent per second (from any node in the network). Note the log scale on the y axis. We start from a steady state system with all pulling periods equal to 0, and one message is sent initially. Nodes receiving the message by the initial push phase generate new messages which result in 198 messages being sent after a warming period of 180 seconds, and a peak of 783 messages once all nodes receive the initial trigger message as they turn to the minimal pulling frequency allowed of one pull request per second.

As there are messages remaining from the burst to be received by nodes in the network, nodes continue to pull with a near-optimal success rate (the average number of useful and useless pull requests are shown on the bottom graph). We observe that nodes stop pulling approximately at the same time. This is an expected result as all nodes are pulling at the same (minimal) period and the large number of messages to be received is much larger than the number of nodes that are reached by the initial push phases. Moreover, these initial push phases reach different set of nodes, allowing all nodes to warm up their pulling activity while distributing initial messages in similar amounts to all of them.

This experiment highlights the ability of Pulp to disseminate at a very low cost (one message per second and per node), large number of messages arriving as burst from multiple senders. We note however that, as claimed in our initial assumptions, Pulp is not tailored nor designed to handle reactive systems (such as eventing mechanisms) where the delays of reception of particular messages are critical. Instead, Pulp carries its goal of disseminating in an adaptive and robust man-

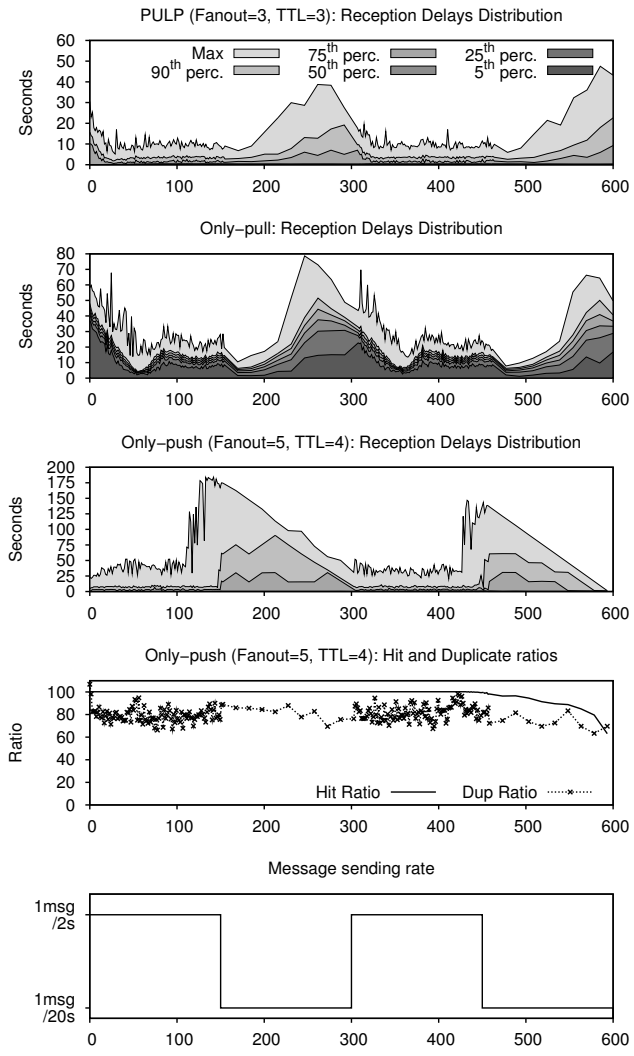

Fig. 10 Comparison of Pulp with an initial seeding phase (top), with no initial seeding phase hence relying only on push operations (middle) and a dissemination that uses only push to disseminate new messages and implicit pull proposals.

ner sets of messages from multiple sources at extremely low cost.

4.5 Comparison to Push-only and Pull-only Disseminations

Our last experiment highlights the benefit of using the Pulp hybrid pull-push approach to dissemination, as opposed to a solution that would use only push or only pull operations. We compare Pulp against two protocols:

– A "pull-only" adaptive protocol. This is basically a subset of Pulp, where the network is not seeded by some initial push phase, but where we keep all other features, in particular, the pulling period dynamic adaptation.
– A "push-only" algorithm, which works as follows. A node that wishes to publish some message initiates a push phase, as for the regular Pulp protocol, but this operation is not completed by regular pull operations. The push itself does not intend to reach a full coverage of the network, which would be totally impractical and overflow any routing infrastructure quickly. Instead, subsequent push operations are used to convey *implicit pull offers* about ongoing disseminations and publicize the availability of new data. When a node n_a pushes some new message m to some node n_b, n_a includes its list of message identifiers (in the same way as in normal Pulp). This list is sorted in the order of message reception as seen by n_a (as all nodes can publish messages, there is no global order on messages, thus two nodes can have different orders for the same set of received messages). In return, n_b requests from n_a the *first* message from n_a's list (the oldest as seen by n_a), among the ones it does not already have. In this way, older messages get larger priorities and, as there are more messages to disseminate, the number of implicit pull offers grows accordingly and helps resolve previous messages.

These two protocols help to highlight the behavior of Pulp itself, as both implement its two key ideas: (1) using pull frequency adaptation to adjust to changes in message emission rates and (2) leveraging a sequence of disseminations from various sources and the spread of information about which elements are missing by an initial push phase.

Figure 10 presents the distribution of dissemination times for a set of 200 messages on a cluster hosting 500 nodes for the three protocols, as well as the duplicate and hit rates of the push-only protocol. Messages are published from random nodes in the network, but according to the frequency that is shown by the bottom plot: alternation between a frequency of 1 messages every 2 seconds and 1 message every 20 seconds, for periods of 150 seconds.

The top plot presents the dissemination times obtained by using Pulp, which are consistent with previous evaluations. Periods of high message sending frequency imply low delays as the number of transient messages in the network is kept to a minimum, yielding more pull messages to sustain the frequency of new messages. Periods of lower sending rate result in lower activity hence higher maximal reception delays, but remaining at acceptable levels with a median delay of at most 10 seconds and a maximal delay of 30 seconds

(note that this can be reduced by setting $\Delta_{\text{pull}_{\max}}$ to a lower value).

The second plot presents the dissemination delay for PULP without the initial push phase. These results convey those from Figure 5: delays increase drastically when the *seeding* effect of the initial push phase is absent. Median delays are effectively doubled when compared to the classical PULP.

The third and fourth plots present the dissemination results for the pure-push scenario: delays distribution, hit ratio (proportion of nodes receiving a message) and duplicate ratio (proportion of useless message reception compared to useful ones). The hit and duplicate ratios are not shown for PULP and its variant with no push phase as the former is always 1 and the latter always 0 by construction. We set the values of TTL and FANOUT for the push-only solution as the minimal ones that yield a hit ratio of 1 for most messages. Under such conditions, the duplicate ratio (mostly due to duplicates during the push phase) is around 80%, that is, a message is received on average 900 times in a network of 500 nodes. The lack of regular pull messages has the consequence that messages are not comprehensively disseminated when no further dissemination takes place, or when the message sending rate drops suddenly.

Overall, this experiment shows that both key components of PULP are necessary for its proper operation, and that, accordingly to the intuition, a hybrid and adaptive approach yields the best results.

5 Related Work

Unlike structured dissemination overlays (typically using trees) that use reactive strategies to tolerate failures and churn, gossip-based approaches proactively implement redundancy in the system and trade network overhead for extreme robustness against failures. Building upon the seminal work in this area where epidemics were introduce to disseminate updates in a database [8], many probabilistic gossip-based approaches have been recently proposed [3, 9, 12, 22, 23]. There has been a growing interest in such proactive approaches in a context where large-scale systems are highly dynamic. They avoid the need for potentially slow recovery mechanisms as well as the cost of maintaining structures in overlay networks. More specifically, many approaches have been proposed in the area of gossip-based video-streaming and secure protocols, breaking the myth of the lack of relevance of gossip protocols to distribute bandwidth-intensive contents or cope with malicious behavior [4, 19, 21, 20, 27, 26].

Gossip-based dissemination protocols usually fall under either of the push-only and pull-only category of protocols as discussed earlier, both having their own tradeoffs with respect to overhead (message redundancy), delay and robustness.

The aforementioned collaborative video streaming protocols combine the two methods in a context-dependent manner: typically push-only protocols are used to disseminate control messages so as for peers to subsequently pull useful stream packets. While the two protocols are combined, they are used for different purposes precisely because they have different characteristics. Push protocols are robust but introduce redundancy and are relevant to disseminate control messages where robustness is required and messages are typically small. Pull protocols are used for the dissemination of large content to limit redundancy. Chainsaw [26] uses pull only to pull for new data in a dynamic network based on a peer sampling mechanism. BAR Gossip [21] and Flightpath [20] focus on tolerating the presence of byzantine peers. In Coolstreaming [19], the content location is pushed while the actual content is pulled as in swarming systems. HEAP [11] accounts for peers heterogeneity by letting nodes dynamically adjust their contribution to gossip dissemination according to their capabilities. In [4], several push-only gossip-based approaches that differ in the choice of the content being pushed are studied. More specifically, the authors demonstrate that sending the most recent chunks to random peers achieves close to optimal dissemination with respect to rate and delay. In [5], a push-only protocol is combined with fountain codes (rateless erasure-correcting codes) to eliminate the unecessary redundancy of standard push protocols.

The approach of PULP is to combine push and pull, not simply to disseminate control messages on one hand and the actual content on the other hand. This approach is shared to some extent by the following work.

The Interleave protocol [29], which is further evaluated by [6], combines push and pull for set of messages as PULP does, but with an approach that differs in several aspects. In Interleave, a list of sequential items is considered. Push is used to propagate new items to a large set of nodes in the system, while pull is used to retrieve the oldest missing items from the set (as decided from the sequence number of newest messages received). As such, Interleave focuses on single-publisher scenarios while PULP targets multiple publishers. Moreover, the frequency of pull operations does not adapt to the frequency of new messages, incurring either a potentially high steady-state load or a lack of reactiveness in periods of heavy publishing activity. On the other hand, an interesting contribution of Interleave is to take into account the bandwidth limitations at each peer, thus supporting more easily high-bandwidth

file diffusion in heterogeneous systems. The properties of push/pull protocols for live video streaming are further studied in [28], with the interesting conclusion that RTT is an essential parameter for such delay sensitive systems (different from those considered by Pulp).

In [25], the authors propose a hybrid dissemination mechanism that also aims at reducing the cost of the push phase by limiting the number of duplicates, and relying on a pull phase thereafter to complete the dissemination. Nonetheless, the main difference with Pulp lies in the approach that is used for limiting the duplicates in that push phase. In [25], a structured network based on prefix routing is used conjunctively with a random network. Both the structured and random overlays are constructed in a delay-aware manner. The structured network is based on a coarse-grain structure, using only a few digits for prefix based routing, allowing a more robust construction and maintenance. The push phase is based on prefix-precedence relations and uses embedded trees that are found in the structured network to *seed* the network with the new messages. Meanwhile, the approach does not allow for frequency-adaptive pull operations. Interestingly, the authors of [25], based on simulation of their protocol, share our observation from Section 4.2, that reaching a small fraction of peers in the initial push phase has only a small increased delay when compared to a push phase that seeds most of the network. This observation pledges in favor of the Pulp approach that stops pushing messages before duplicates are likely to occur rather than maintaining a structure amongst peers to ensure this property, as the former is more lightweight and robust in the long term. In any case, the two protocols focus on optimizing different metrics: [25] concentrates on minimizing delays, while Pulp prioritizes on lowering traffic, be it redundant data transfers or control messages.

In [14], Karp *et al.* theoretically analyze the combination of push and pull for disseminating a *single* message. They propose using push-based dissemination until $n/\log n$ nodes have the message with high probability (*exponential growth* phase). From that point on, each uninformed node has sufficiently high probability of reaching an informed node by probing nodes at random, so they switch to pull communication (*quadratic shrinking* phase). The overall message complexity is $\mathcal{O}(n \log \log n)$. As the algorithm relies on an exact estimation of the number of rounds to execute during the first phase, Karp *et al.* propose a *median-counter* algorithm to determine the right phase transition time. Nonetheless, contrary to Pulp that strives to avoid duplicate message deliveries, the goal of this mechanism is to reduce delays. Hence, the first phase (push) is used for as long as the probability of hitting a non-informed node is higher than the probability of a non-informed node randomly probing an informed one. As a side-effect of limiting the push phase, the number of redundant deliveries is reduced to some extent, however, it is far from being eliminated. Moreover, Karp *et al.* focus on the independent dissemination of *individual messages*, not taking advantage of streams of messages and the potential interplay these messages can have in enabling faster or more reliable dissemination overall.

Liu *et al.* [24] also use a hybrid mechanism that mixes push and pull interactions for cache replica maintenance. In this context, a push operation refers to a proactive replication from the "master" node and a pull operation to a passive replication from nodes holding replicas to nodes with free space as a result of periodic exchanges.

Also in the domain of cache updates, Srinivasan *et al.* [30] and Urgaonkar *et al.* [31] proposed to dynamically adapt the frequency of pull requests from replica holders to master nodes for the dynamic update of read-only replicas in caches. Adaptation is performed in a similar manner as in Pulp, by adding a constant to the frequency as the number of updates to replicas increases, and dividing the frequency when experiencing too many useless pull requests.

6 Conclusion

The properties of gossip-based protocols have been widely studied in the literature. Such protocols rely on randomization and are known to be simple, scalable and extremely robust. Yet, they have long been deemed impractical, mainly because of the large gap between their behaviors as predicted by theoretical models and as experienced in real networks. More specifically, the correct operation of gossip-based protocols depends on many external factors that are difficult to dimension properly without good knowledge of the underlying system (e.g., its size, delays, lossiness, etc.). Moreover, gossip-based dissemination protocols are usually considered much more expensive than deterministic protocols in terms of overhead. Yet, the robustness to churn of such protocols make them more and more appealing to the point that they have recently been used in the context of video streaming applications.

In this context, we have presented Pulp, a lightweight gossip-based dissemination protocol that combines pull- and push-based approaches and performs remarkably well in practice. Pulp disseminates flows of messages originating from multiple sources to large sets of nodes in a fully decentralized way. Gossip messages exchanged by the push protocol carry information that

helps nodes perform pulls in a smart and self-adaptive manner.

Thanks to its hybrid approach, PULP limits the redundant traffic from the push phase and the unnecessary polling from the pull phase, thereby being particularly network-efficient. Our deployment of PULP in real-world conditions on a cluster and on PlanetLab demonstrates its good performance and its robustness even in the face of high churn. While PULP seamlessly supports node failures, it does not explicitly take into account the presence of selfish or malicious nodes. This is an interesting area of research left for future work.

Acknowledgements This work was carried out during the tenure of Étienne Rivière's ERCIM (European Research Consortium for Informatics and Mathematics) "Alain Bensoussan" fellowship. This work is supported in part by the Swiss National Foundation Grant 102819.

References

1. http://www.planet-lab.org/.
2. Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 256–267, Berkeley, CA, USA, February 2003.
3. Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Computer Systems*, 17(2):41–88, May 1999.
4. Thomas Bonald, Laurent Massoulié, Fabien Mathieu, Diego Perino, and Andrew Twigg. Epidemic live streaming: optimal performance trade-offs. In *SIGMETRICS*, pages 325–336, Annapolis, MA, USA, June 2008.
5. Mary-Luc Champel, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Fog: Fighting the achilles' heel of gossip protocols with fountain codes. In *SSS*, pages 180–194, Lyon, France, 2009.
6. Renato Lo Cigno, Alessandro Russo, and Damiano Carra. On some fundamental properties of P2P push/pull protocols. In *Proceedings of the 2nd International Conference on Communications and Electronic (HUT-ICCE)*, pages 67–73, HoiAn, Vietnam, jun 2008.
7. B. Cohen. Incentives to build robustness in BitTorrent. Technical report, `http://www.bittorrent.com/bittorrentecon.pdf`, May 2003.
8. Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, Canada, August 1987.
9. Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov. Lightweight probabilistic broadcast. *ACM Transaction on Computer Systems*, 21(4), November 2003.
10. Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Boris Koldehofe, Martin Mogensen, Maxime Monod, and Vivien Quéma. Heterogeneous gossip. In *Middleware*, Urbana, IL, USA, 2009.
11. Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Martin Mogensen, Maxime Monod, and Vivien Quéma. Gossiping Capabilities. Technical Report LPD-REPORT-2008-010, EPFL, 2008.
12. Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, February 2003.
13. Mark Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), August 2007.
14. R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumour spreading. In *IEEE Proc. 41st Ann. Symp. Foundations of Computer Science (FOCS)*, page 565, November 2000.
15. Srinivas Kashyap, Supratim Deb, K. V. M. Naidu, Rajeev Rastogi, and Anand Srinivasan. Efficient gossip-based aggregate computation. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 308–317, New York, NY, USA, June 2006. ACM Press.
16. Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):139–149, March 2003.
17. Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *ACM Operating System Review*, 41(5):2–7, October 2007.
18. Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Kenneth P. Birman, and Alan J. Demers. Active and passive techniques for group size estimation in large-scale and dynamic distributed systems. *Journal of Systems and Software*, 80(10):1639–1658, 2007.
19. Bo Li, Yang Qu, G.Y. Keung, Susu Xie, Chuang Lin, Jiangchuan Liu, and Xinyan Zhang. Inside the new coolstreaming: Principles, measurements and performance implications. In *Proceedings of the 27th Conference on Computer Communications (IEEE INFOCOM)*, pages 1031–1039, April 2008.
20. Harry C. Li, Allen Clement, Mirco Marchetti, Manos Kapritsos, Luke Robison, Lorenzo Alvisi, and Mike Dahlin. Flightpath: Obedience vs. choice in cooperative services. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, December 2008.
21. Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Lorenzo Alvisi, and Michael Dahlin. BAR gossip. In *Proc. of 7th Symposium on Operating System Design and Implementation (OSDI '06)*, pages 191–2004, November 2006.
22. Meng Lin and Keith Marzullo. Directional gossip: Gossip in a wide area network. Technical Report CS1999-0622, University of California, San Diego, 1999.
23. Meng Lin, Keith Marzullo, and Stefano Masini. Gossip versus deterministic flooding: low-message overhead and high-reliability for broadcasting on small networks. In *Intl. Symposium on Distributed Computing (DISC 2000)*, pages 85–89, Toledo, Spain, 2000.
24. Xiaotao Liu, Jiang Lan, Prashant Shenoy, and Krithi Ramaritham. Consistency maintenance in dynamic peer-to-peer overlay networks. *Computer Networks*, 50(6):859–876, 2006.
25. Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. Push-to-pull peer-to-peer live streaming. In *Proceedings of DISC 2007: 21st International Symposium on Distributed Computing*, Lemosos, Cyprus, sep 2007.
26. Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *IPTPS'05: the fourth International Workshop on Peer-to-Peer Systems*, pages 127–140, February 2005.

27. Fabio Picconi and Laurent Massoulié. Is there a future for mesh-based live video streaming? In *Proceedings of the Eigth International Conference on Peer-to-Peer Computing (P2P'08)*, Aachen, Germany, September 2008.
28. Alessandro Russo and Renato Lo Cigno. Delay-aware push/pull protocols for live video streaming in p2p systems. In *Proceedings of the IEEE International Conference on Communications (ICC)*, May 2010.
29. Sujay Sanghavi, Bruce Hajek, and Laurent Massoulié. Gossiping with multiple messages. In *Proceedings of the 29th Conference on Computer Communications (IEEE INFOCOM)*, pages 2135–2143, May 2007.
30. Raghav Srinivasan, Chao Liang, and Krithi Ramamritham. Maintaining temporal coherency of virtual data warehouses. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 60, Washington, DC, USA, 1998. IEEE Computer Society.
31. Bhuvan Urgaonkar, Anoop George Ninan, Mohammad Salimullah Raunak, Prashant Shenoy, and Krithi Ramamritham. Maintaining mutual consistency for cached web objects. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 371, Washington, DC, USA, April 2001. IEEE Computer Society.
32. Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In IFIP, editor, *Proc. of Middleware, the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 55–70, The Lake District, UK, 1998.
33. Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2), June 2005.
34. X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. Coolstreaming/DONet: A data-driven overlay network for efficient live media streaming. In *Proceedings of the 24th Conference on Computer Communications (IEEE INFOCOM)*, pages 2102–2111, March 2005.

Pascal Felber received his M.Sc. and Ph.D. degrees in Computer Science from the Swiss Federal Institute of Technology. He has then worked at Oracle Corporation and Bell-Labs in the USA, and at Institut EURECOM in France. Since October 2004, he is a Professor of Computer Science at the University of Neuchâtel, Switzerland, working in the field of dependable, distributed, and concurrent systems. He has published over 80 research papers in various journals and conferences.

Anne-Marie Kermarrec is a Senior Researcher at INRIA (France) since 2004, where she is leading the ASAP (As Scalable As Possible) research group, focusing on large-scale dynamic distributed systems. Her research interests are peer to peer networks, large-scale information management and epidemic protocols. Before that, Anne-Marie was with Microsoft Research in Cambridge (UK). She obtained her Ph.D. from the University of Rennes (France) in October 1996. Anne-Marie has been awarded a European Research Council Starting Grant in 2008 for her 5 year GOSSPLE project.

Lorenzo Leonini graduated in computer science in 2005. Since then, he has been a PhD student at the university of Neuchâtel, where his research interest have been in the design and evaluation of large-scale distributed systems. Lorenzo has been working on distributed systems for information retrieval, epidemic dissemination, recommendation systems, amongst others. He is the main designer and architect of the Splay system.

Étienne Rivière is a lecturer at the University of Neuchâtel, Switzerland. He received his PhD in Computer Science from the University of Rennes, France in November 2007. In 2008 and 2009, Étienne has been a postdoctoral fellow under an "Alain Bensoussan" grant from ERCIM, which led him to the University of Neuchâtel and to NTNU Trondheim in Norway. His research interests lie in the design, analysis, implementation and evaluation of large-scale distributed systems and concurrent systems. He is a member of the ACM, the IEEE and Usenix.

Spyros Voulgaris is an Assistant Professor at the Vrije Universiteit Amsterdam since October 2008, focusing on massive scale decentralized systems both in wired and wireless settings. Before that, he was a postdoc at ETH Zurich since 2006. He has obtained his Ph.D. from the Vrije Universiteit Amsterdam (2006), his M.Sc. from the University of Michigan in Ann Arbor (1999), and his B.Sc. from the University of Patras (1997). He has worked at Microsoft Research Cambridge (2003), Hughes Network Systems in Maryland (1999-2001), and HP Labs (1998).