ElastO: Dynamic, Efficient, and Robust Maintenance of Low Fan-out Overlays for Topic-based Publish/Subscribe under Churn

Chen Chen[1], Roman Vitenberg[2], and Hans-Arno Jacobsen[1]

[1] Department of Electrical and Computer Engineering, University of Toronto
[2] Department of Informatics, University of Oslo, Norway
{chenchen,jacobsen}@eecg.toronto.edu, romanvi@ifi.uio.no

Abstract. We propose, ElastO, a distributed system for constructing and maintaining scalable churn-resistant overlay networks for topic-based publish/subscribe (pub/sub) systems. ElastO is designed to dynamically tread the balance among several key dimensions: (a) topic-connected overlay (TCO), i.e., the sub-overlay induced by nodes interested in any topic is connected, (b) low maximum and average node fan-outs, (c) high efficiency to maintain the overlay in presence of churn, (d) balanced computation and communication overhead across all the nodes.
Existing approaches for maintaining pub/sub TCOs are either static and runtime-costly algorithms, or decentralized protocols that produce significantly higher node degrees. One main challenge is to effectively overcome departure of nodes central to the TCO. ElastO carefully maintains *local view* at each node and efficiently computes sets of *shadow nodes* upon churn events, so that all links adjacent to a failed node can be quickly replaced by adding links among the shadow nodes.
We evaluate ElastO using both synthetical pub/sub workloads and practical workloads extracted from Facebook and Twitter, and using real-world cluster churn traces released by Google. We show analytically and experimentally that ElastO achieves low fan-out close to static algorithms and high efficiency comparable to decentralized protocols.

1 Introduction

A distributed topic-based publish/subscribe (pub/sub) system often organizes nodes (e.g., brokers, servers or routers) in a federated or peer-to-peer manner as an overlay at the application or network layer. The properties of the overlay directly impact the performance and scalability of the pub/sub system, such as the message routing cost. Constructing a high-quality broker overlay is a fundamental problem for distributed pub/sub that has received attention both in industry [23] and academia [6,20,25,7,22].

From the practical perspective, the innate dynamism of networks requires distributed pub/sub to be capable of maintaining the overlay in presence of churn. The pub/sub system may need to add or upgrade brokers to accommodate the increasing load from time to time. Nodes may depart the system due to failures or administrative maintenance. In practice, the composition of machines in a

data center has non-negligible variation over time [1]. Furthermore, the advent of new pub/sub applications, e.g., in sensor networks or mobile networks, makes it increasingly important and challenging to tackle churn while maintaining the overlay. In these applications, the overlay nodes are not necessarily dedicated brokers – they could be cell phones, cameras or laptops, and often operate in moving environment. These devices have limited resources (e.g., memory and bandwidth), are failure-prone, and need to be switched periodically to stand-by mode for power saving. Thus pub/sub is subjected to increasing dynamism and induces additional resource constraints and real-time requirements.

We present ElastO for maintaining high-quality message dissemination overlays for topic-based pub/sub in presence of churn. The system properties include:

A. *Topic-connected overlay (TCO)*, which informally speaking, means that all nodes interested in the same topic are organized in a connected dissemination sub-overlay [6]. This concept is applicable to both peer-to-peer solutions for pub/sub in which the clients form the overlay and broker-based solutions in which the brokers form the overlay. This abstraction does not differentiate between publishers and subscribers, because it simplifies the presentation for a theoretical and algorithmic treatment of the problem, while fully preserving its practical character. This property ensures that nodes not interested in a topic never need to contribute to disseminating information on that topic. Publication routing atop such overlays saves bandwidth and computational resources otherwise wasted on forwarding messages of no interest to the node. It also results in simple routing protocols and smaller forwarding tables. From a security perspective, topic-connectivity is desirable when messages are to be shared across a network among a set of trusted users without leaving this set.

B. *Low fan-out*: which ensures that both maximum and average degrees are small. It is imperative for a pub/sub overlay to have low fan-out because it costs a lot of resources to maintain adjacent links for a high-degree node (i.e., monitor the links and the neighbors [6,20]). Furthermore, for a typical pub/sub system, each link would have to accommodate a number of protocols, service components, message queues, etc. While overlay designs for different applications might be principally different, they all strive to maintain bounded node degrees, e.g., DHTs [18] and peer-to-peer streaming [2].

C. *Churn resistance*: It is important to restore TCO upon node departure as soon as possible by pre-computing the knowledge before the churn occurs. In particular, the existing algorithms for building a high-quality TCO from scratch are known to be computationally expensive. It is imperative to avoid carrying this runtime cost into a dynamic solution.

D. *Balanced load* of computation and communication due to overlay maintenance across all nodes in the overlay. Both restoring topic-connectivity and monitoring the overlay during the normal operation incurs communication as well as computation overhead. It is important to spread this overhead fairly and evenly across the nodes in a distributed system.

To the best of our knowledge, none of the approaches in the state-of-the-art manage to satisfy all of the above mentioned properties at the same time. We can classify existing works into two categories (see Table 1): (a) centralized algo-

rithms that *statically* construct a provably low-degree TCO from scratch and (b) decentralized protocols that strive to *dynamically* maintain low degrees (and in many cases, topic-connected) in a best-effort fashion. Unfortunately, the former are known to have high runtime cost, which makes them unsuitable as a dynamic solution. Meanwhile, the latter produce significantly higher node degrees compared to the former – e.g., the node degrees produced by PolderCast [25] grow almost linearly with subscription size under typical pub/sub workloads.

Table 1: Approaches to construct pub/sub TCO

		Knowledge	Churn Handling	Runtime	Avg Degree	Max Degree
ElastO		Local	✓	Fast	$\approx O(\rho \log \lvert V\rvert \lvert T\rvert)$	$\approx O(\frac{\lvert V\rvert}{\rho} \log \lvert V\rvert \lvert T\rvert)$
Centralized Algorithms	LowODA [20]	Global	✗	Slow	$O(\rho \log \lvert V\rvert \lvert T\rvert)$	$O(\frac{\lvert V\rvert}{\rho} \log \lvert V\rvert \lvert T\rvert)$
	GM [6]	Global	✗	Slow	$O(\log \lvert V\rvert \lvert T\rvert)$	$\Theta(\lvert V\rvert)$
	MinMaxODA [20]	Global	✗	Slow	$\Theta(\lvert V\rvert)$	$O(\rho \log \lvert V\rvert \lvert T\rvert)$
Decentralized Protocols	[7,22,25]	Global/Local	✓	Fast	Unknown	Unknown

In contrast, we propose ElastO, a *hybrid* approach between centralized algorithms and decentralized protocols, which combines the strengths from both. We summarize the contributions in this work as follows:

I. The ElastO approach is *the first system* with *complete architecture and protocol design* (see §4) such that (1) TCO is guaranteed to be quickly restored (in the same order of magnitude as the most recent decentralized protocols), and (2) the node degrees stay provably low with approximation ratios that slightly exceed the ones for the static baseline algorithms, while the overlay is dynamically evolving under incremental node churn.

II. We propose novel TCO repairing algorithms with *shadow sets* in §5. The main obstacle of dynamic maintenance is to handle the departure of a node central to TCO. In this situation, additional edges need to be created in order to mend the overlay and restore topic-connectivity. If the system considers the entire set of potential edges that can be added, its running time will be problematic. If the system only considers a small subset of edges for addition, those edges may turn out suboptimal thereby significantly increasing the node degrees. To overcome these challenges, ElastO calculates a *shadow set* of nodes upon each churn event, and then compute edges among the shadow set for restoring TCO.

III. We design a unique *local view selection mechanism* in §6. ElastO continuously maintains a local view at each node, which includes a set of *backup nodes* to support the TCO repairing algorithms with shadows. We construct the backup set when a node joins and incrementally updated if needed, e.g., when some nodes become unavailable. When node v' leaves, we can efficiently obtain the shadow set for this churn event, which is the union of the neighbors of v' and the backup set for v'. We compute the backup set in such a way that (a) the computation and communication overhead of incrementally restoring topic-connectivity is low, (b) the degrees are kept provably low, (c) the backup sets can be efficiently updated, and (d) the backups should be well-balanced distributed among all the nodes in the system, any node does not serve as a backup for a large number

of other nodes. These requirements imply the trade-offs with regard to the size of the backup set, which is key to tune the degree of decentralization in ElastO: if the backup set is of a larger size, then the local view and the shadow set become closer to the global view, and thus ElastO becomes less decentralized and more centralized – the output TCO gains higher quality at the cost of more expensive costs in both maintaining local view and repairing TCO.

IV. We conduct comprehensive experiments in §7 under various a variety of pub/sub workloads, including both synthetically generated workloads and large-scale real-world workloads (i.e., Facebook [27] and Twitter [15]) with up to 10K nodes, 10K topics and 5K subscriptions per-node. We evaluate the properties of ElastO through a month-long period churn trace from Google cluster data [1].

2 Related Work

A significant body of research has been considering the construction of an overlay topology underlying pub/sub systems such that the network traffic is minimized (e.g., [6,20]). TCO is explicitly required in [7,22,25] and implicitly preferred in [4,3], which all aim to reduce the number of intermediate overlay hops for a message to travel in the network.

On the one hand, aiming to achieve TCO while minimizing node degrees has been explored algorithmically [6,20]. Different optimization goals lead to a number of polynomial-time algorithms. Those algorithms have proven approximation bounds in the worst-case scenarios and can serve as comparison baselines for developing other approaches.

Unfortunately, all the above state-of-the-art algorithms are static by design: (1) requiring global knowledge, (2) assuming centralized operation, and (3) constructing the overlay from scratch only. Because of these innate static properties the algorithms do not lend themselves to dynamic environments. In particular, the state-of-the-art algorithms suffer from the high runtime complexity so that it is impractical to rerun them each time a single node joins or leaves. Furthermore, it is also prohibitively expensive to tear down a large number of existing links and to re-establish different new links.

On the other hand, systems like [7,22,25] build the TCO in a decentralized manner. These systems implement non-coordinated decentralized overlay construction protocols such that each node decides upon its own neighbors. The protocols are fast and they can operate with only partial knowledge.

However, these heuristics do not provide any theoretical guarantees for the node degrees in these systems. In practice, the node degrees are usually multiple times higher than the bounds provided by the static baselines [6,20].

In contrast, we adopt a principally different approach that resides between the static algorithms and decentralized protocols. In ElastO, only a subset of nodes, which we call shadow set, are involved in the automatic overlay recovery when churn occurs. The conceptual idea of shadow set also appears in [17], which, however, is not designed for pub/sub communication and is nontrivial to address the new topological properties required by pub/sub. We also tailer recent peer sampling services [13,14] for our own purposes and rely on existing failure detectors [9] as a building block.

3 Background

Let $I(V, T, Int)$ represent an input instance, where V is the set of nodes, T is the set of topics, and Int is the interest function such that $Int : V \times T \rightarrow \{true, false\}$. Since the domain of the interest function is a Cartesian product, we also refer to this function as an interest matrix. Given an interest function Int, we say that a node v is interested in some topic t if and only if $Int(v, t) = true$. We also say that node v subscribes to topic t.

We denote by $TPSO(V, T, Int, E)$ a *topic-based pub/sub overlay network*. A $TPSO(V, T, Int, E)$ can be illustrated as an undirected graph $G = (V, E)$ over the node set V with the edge set $E \subseteq V \times V$. Given $TPSO(V, T, Int, E)$, the sub-overlay *induced* by $t \in T$ is a subgraph $G^{(t)} = (V^{(t)}, E^{(t)})$ such that $V^{(t)} = \{v \in V | Int(v, t)\}$ and $E^{(t)} = \{(v, w) \in E | v \in V^{(t)} \wedge w \in V^{(t)}\}$. A *topic-connected component* (*TC-component*) on topic $t \in T$, is a maximal connected subgraph in $G^{(t)}$. A *TPSO* is a *topic-connected overlay* if for each topic $t \in T$, $G^{(t)}$ has at most one *TC-component*, and we denote it as $TCO(V, T, Int, E)$.

4 The ElastO Approach

4.1 Scopes of ElastO

We lay out the practical application scenarios that guide our system design:

- The system is built from many inexpensive commodity components, and a small but non-negligible number of server and network components can fail at any given time. The system must constantly monitor itself, detect, tolerate, and recover promptly from failures on a routine basis [10,16].
- The system needs to add nodes from time to time. For example, previously failed nodes can rejoin the network after being fixed, or the system may require new nodes to accommodate the increasing load [16].
- The target environment is large-scale data center with moderate churn [16]. First, most of the churn are *simple*, i.e., only one node joins, leaves, or fails at a time. Second, the intervals between successive churn rounds are in the order of tens of minutes, depending on the size of the cluster [1]. Concurrent churn events involving multiple nodes occur infrequently.

4.2 Overview

ElastO resides between the pub/sub routing protocols layer and the network transport protocols layer. The major functionality of ElastO is constructing and maintaining pub/sub TCO (i.e., the message dissemination overlay for pub/sub) in the presence of churn – how new nodes join the overlay, how to recover the TCO from the failure (or planned departure) of existing nodes, etc. In particular, ElastO needs to have the following functionalities: (a) membership management, (b) failure detection, (c) TCO recovery, and (d) distributing the responsibilities for TCO recovery across the nodes in a balanced fashion.

Based on the observations listed in §4.1, we summarize the design principles:

1. The system allows any node to join or leave at any time. There are no dedicated servers, and the system needs to be highly scalable and available to support continuous growth.

2. Any node may fail at any time, and the system treats failures as the norm rather than the exception. We concentrate on node crashes in the failure model. In particular, we do not consider link failures or Byzantine failures.

3. ElastO supports correctness for concurrent churn events involving up to L nodes simultaneously (L is a configuration parameter). Without loss of generality, concurrent churn results in a sequence of simple churn events produced by the failure detector. For each simply churn event only one node joins, leaves, or fails. The next churn event may occur before the handling of the previous event is completed. Even in this case, topic-connectivity is eventually restored in the overlay. However, such concurrent churn may result in a small number of redundant messages being sent and a number of redundant links created in the overlay. This is justified by our assumptions in §4.1 and furthermore, evaluated in the experiments described in §7.

4. When a node w departs from the overlay, the departure may render the overlay disconnected for many topics. The overlay is repaired by creating additional links between a set of nodes specially designated as *backup nodes* for w (each node has a set of designated backups assigned to it). If the backup set is large, the repair becomes similar to centralized static algorithms. If the backup set is of small cardinality, the system leans more towards decentralized protocols. A principal contribution of our design is in maintaining the balance when it comes to the backup set size: we show that it is possible to keep the size moderate, so that the computation is cheap as in decentralized protocols, yet sufficient for the overlay quality to approach that produced by centralized algorithms. We discuss the choice of the backup set and related trade-offs §5.

5. The crux of our design is in the even distribution of (a) backup nodes, (b) responsibility for repairing the overlay, and (c) load due to maintaining meta-information in a decentralized fashion.

Alg. 1 Basic data structures maintained by each node $\mathbf{v} \in \mathbf{V}$

NODE: encapsulation of node descriptor
- *id*: node identifier
- *topics*: subscribed topics
- *neighbIds*: nodeIds of TCO neighbors

CHURNEVT: simple churn event with a node
- *type*: churn type, JOIN or LEAVE
- *node*: NODE object for the churn node

MESSAGE: abstract class to encapsulate messages transmitted between two nodes
- *source*: NODE of message source
- *mclass*: message class. Each class extends abstract MESSAGE to a concrete subclass.

Alg. 1 specifies basic data structures in ElastO.

1. NODE defines the *node descriptor*, which includes the identities of nodes along with their metadata, including topic interests and TCO neighbors. We use NODE as elementary entry for many local container variables maintained at each node, e.g., $v.\mathscr{N}$, $v.\mathscr{D}$, and $v.\mathscr{L}$ in Alg. 3.

2. CHURNEVT represents the class of simple churn events with only one node. Each CHURNEVT object needs to instantiate the churn node descriptor in CHURNEVT.*node* and the churn type in CHURNEVT.*type*, e.g., JOIN or LEAVE.

3. MESSAGE is abstraction for messages transmitted between nodes.

Alg. 2 extends base MESSAGE to different concrete subclasses for various purposes of the protocols. Alg. 3 presents key local variables at each node and the

framework of ElastO protocols. Each node $v \in V$ maintains a set of node descriptors, $v.\mathcal{N}$, as the overlay neighbors. The $v.\mathcal{N}$ over all $v \in V$ define the ElastO pub/sub overlay. We denote by $v.\mathcal{D}$ the updated output list of the failure detector at node $v \in V$, and node v can query the local failure detector by checking whether $w \in v.\mathcal{D}$ at any time. Each node $v \in V$ also keeps a *local view* $v.\mathcal{L}$ about other nodes in the network. The view includes node descriptions for L successors and predecessors of v in the circular space, the list of neighbors (node descriptors) for predecessors and most importantly, the list of backup nodes.

Alg. 2 Extended subclasses of MESSAGE

- ▷ *RepairQ*: request to repair TCO
  - ∘ *chnevt*: the associated CHURNEVT
- ▷ *RepairS*: TCO repair response
- ▷ *AddneiQ*: request to add new neighbor
  - ∘ *newnei*: new TCO neighbor
- ▷ *AddneiS*: new neighbor response
- ▷ *NeighbQ*: request about TCO neighbors
  - ∘ *nodeId*: the nodeId to request
- ▷ *NeighbS*: neighbor update response
  - ∘ *neighbs*: TCO neighbor descriptors
- ▷ *LvQ*/*LvS*: local view request/response
  - ∘ *view*: view descriptors to exchange

Alg. 3 ElastO Local Protocol for Message/Event Handling at Node $\mathbf{v} \in \mathbf{V}$

// **v**'s field attributes and local variables

▷ $\mathbf{v}.\mathcal{N}$: overlay neighbors
▷ $\mathbf{v}.\mathcal{D}$: failed nodes, output of the failure detector
▷ $\mathbf{v}.self$: self descriptor
▷ $\mathbf{v}.succ[l], 0 \le l < L$: successor descriptors
▷ $\mathbf{v}.pred[l], 0 \le l < L$: predecessor descriptors
▷ $\mathbf{v}.pneighb[l], 0 \le l < L$:
neighbor descriptors of the predecessors
▷ $\mathbf{v}.backup[l], 0 \le l < L$:
backup descriptors for the predecessors
▷ $\mathbf{v}.\mathcal{L}$: local view, set union of $\mathbf{v}.self$, $\mathbf{v}.succ[l]$, $\mathbf{v}.pred[l]$, $\mathbf{v}.pneighb[l]$, $\mathbf{v}.backup[l]$, $\forall l, 0 \le l < L$

// **v**'s local functions

► **v**.handleEvent(*chnevt*)
// upon detecting *chnevt* locally
1: repairTcoOnChurn(*chnevt*)

► **v**.handleMessage(*msg*)
// upon receiving *msg* from some node
1: **switch** (*msg.mclass*)
2: **case** *RepairQ*: handleRepairQ(*msg*)
3: **case** *AddneiQ*: handleAddneiQ(*msg*)
4: **case** *NeighbQ*: handleNeighbQ(*msg*)
5: **case** *LvQ* or *LvS*: handleLvQS(*msg*)

► **v**.handleRepairQ(*msg*)
1: **if** *msg.chnevt* captured before **then**
2: **wait until** *msg.chnevt* handled
3: **else**
4: repairTcoOnChurn(*msg.chnevt*)
5: send(*msg.source*, **new** *RepairS*(**v**.*self*))

► **v**.handleAddneiQ(*msg*)
1: $w \leftarrow msg.newnei$
2: **if** $w \notin \mathbf{v}.\mathcal{D}$ **then**
3: $\mathbf{v}.\mathcal{N} = \mathbf{v}.\mathcal{N} \cup \{w\}$
4: send(*msg.source*, **new** *AddneiS*(**v**.*self*))

► **v**.handleNeighbQ(*msg*)
1: *msgRes* ← **new** *NeighbS*(**v**.*self*)
2: **if** $msg.nodeId = \mathbf{v}.self.id$ **then**
3: $msgRes.neighbs \leftarrow \mathbf{v}.\mathcal{N}$
4: **else**
5: $l \leftarrow$ find l s.t. $\mathbf{v}.pred[l].id = msg.nodeId$
6: $msgRes.neighbs \leftarrow \mathbf{v}.pneighb[l]$
7: send(*msg.source*, *msgRes*)

► **v**.handleLvQS(*msg*)
1: **if** *msg.mclass* is *LvQ* **then**
2: *msgRes* ← **new** *LvS*(**v**.*self*, $\mathbf{v}.\mathcal{L}$)
3: send(*msg.source*, *msgRes*)
4: *buffer* ← $\mathbf{v}.\mathcal{L} \cup msg.view$
5: selectLV(*buffer*)

We implement a peer sampling service to build the local view for each node. The service uses a data exchange protocol similar to those in [13,14]. Yet, the information included for each node in the view is much broader and our local view selection mechanism is unique (see §6), especially because we rely on the local view for repairing TCO (see §5).

One of the key design features for ElastO is the ability to scale dynamically and incrementally. This requires ElastO to evenly distribute churn handling overhead over all nodes in the network. We design a "primary-backup" strategy to balance these loads based on an *identifier circle*. ElastO randomly assigns each node an unique m-bit node identifier (nodeId), which represents its position in an *identifier circle* module 2^m. The identifier circle places a linear ordering over all nodeIds in a circular space, and each ElastO node has *predecessors* and *successors* on the identifier circle. We can generate nodeIds by using consistent hashing [24] on each node's public key or its IP address.

The identifier circle allows ElastO to delegate the responsibility of handling each simple churn event to a specific node, depending on the location of the churn node. Once a churn event occurs (e.g., node v' joins or leaves), the immediate successor to the churn node v', say node v, is deemed the *churn coordinator*, i.e., node v is responsible for repairing the TCO upon the churn event at v'. As Fig. 1 shows: when node v_{10} failed, its immediate successor v_{11} captured this churn event and worked as the *churn coordinator* to recover the TCO.

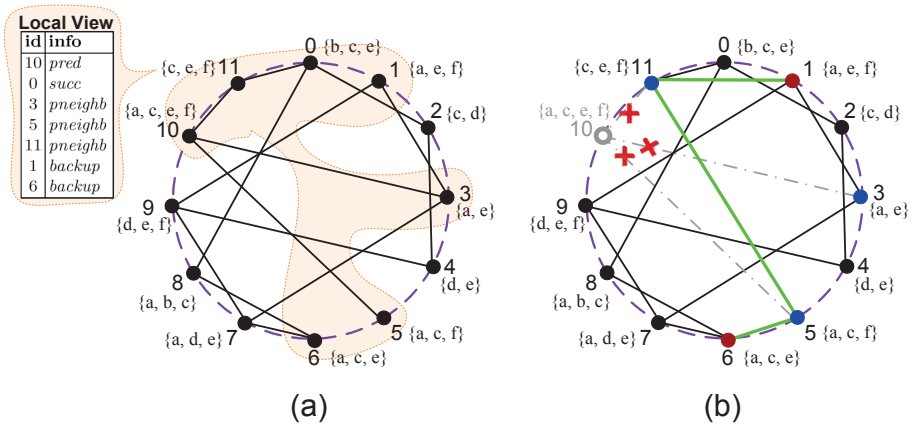


Fig. 1: Example of the ElastO for pub/sub. **(a)** An initial TCO on top of the circular nodeId space, where each node $v \in V$ keeps a $v.\mathscr{L}$ locally. **(b)** Node v_{10} is leaving from the overlay, and the churn coordinator v_{11} repairs the TCO by adding edges $\{(v_{11}, v_1), (v_{11}, v_5), (v_5, v_6)\}$ among $S = \{v_{11}, v_1, v_3, v_5, v_6\}$, a subset of $v_{11}.\mathscr{L}$.

Coordinator v reacts to churn event by invoking TCO repairing algorithms based on its local view $v.\mathscr{L}$. It is also possible that the coordinator v is not ready for handling the incoming churn event, i.e., the peer sampling procedure has not built up necessary knowledge for $v.\mathscr{L}$. However, this situation rarely happens, only when v is a newly joined node, the number of tolerated concurrent churn events $L > 1$, and v' leaves before v knows about the neighbors of v'. In that case, v can retrieve the necessary information from its successors.

If some other node w (rather than v''s immediate successor) detects the churn event about v', node w would create a ChurnEvt object, wrap the object into a *RepairQ* message, and then forward the message to v''s closest successor node chosen from $w.\mathscr{L}$. Because w's knowledge about the successor of v' might be imprecise, it is possible that the node to which w sends a a *RepairQ* message is not the actual successor, in which case it needs to forward the message further.

The constructed ChurnEvt event will be propagated recursively until it reaches node v, which invokes the actual operations for churn handling. This forwarding process resembles the routing for locating a key in DHT [24], but we do not use an existing DHT because our local view has different requirements as compared to the DHT routing table (see §6).

To handle other corner cases, we adopt a relaxed consistency model that facilitates efficient churn handling:

(1) We do not guarantee the consistency for the local view among all $v \in V$. Suppose both v and w keeps u's node descriptor in its local view, denoted as u_v and u_w, respectively, then it is possible that $u_v.neighbIds \neq u_w.neighbIds$. Either $u_v.neighbIds$ or $u_w.neighbIds$ does not have to reflect node u's real TCO neighbor set. However, we do guarantee that $u_v.neighbIds$ (or $u_w.neighbIds$) always corresponds to a subset u's TCO neighbors, i.e., $u_v.neighbIds$ might be incomplete, but it never contains a member that is not u's neighbor.

(2) We do not impose consensus about the churn coordinator selection among all nodes upon churn. However, we ensure that at least one node takes charge of each churn, Although it is possible that more than one node is assigned as the churn coordinator at the same time, it does not impact the topic-connectivity of the overlay – only introducing some redundant links. We can add some garbage collection mechanism to improve the optimality of the TCO in a lazy manner.

(3) Node v may receive a number of TCO repair requests from multiple nodes that detect the churn simultaneously, but v only invokes the TCO repairing algorithm once for the same churn event.

To fulfill these functionalities, we design ElastO with three major modules:

I. MemberServ maintains the identifier circle and keeps the *local view* updated continuously at each node. It is also responsible for detecting failures.

II. TcoUpgrader repairs the pub/sub TCO upon churn. It implements TCO construction algorithms for the churn coordinator.

III. TcoCreator constructs the TCO from scratch initially (or periodically once in a while). TcoCreator can produce a close-to-optimal TCO, but it admits centralized operations, requires the global knowledge, and thus imposes expensive computation. The system invokes TcoCreator very infrequently.

We introduce each architectural module in §4.4, §4.3, and §4.5, respectively. Then §5 discusses the overlay repairing algorithms in TcoUpgrader, and §6 explains the local view selection mechanism in MemberServ.

4.3 The Membership Service Module

To maintain the local view, MemberServ extends a gossip-based peer sampling service [13,14]. Each node v periodically exchanges its $v.\mathscr{L}$ with another node w, chosen uniformly among the existing members in $v.\mathscr{L}$. Node v, then, merges its current $v.\mathscr{L}$ with $w.\mathscr{L}$ as a fresh list of local view candidates. Next, v chooses a number of members among the candidates according to some selection criteria and updates $v.\mathscr{L}$. The same process takes place at node w (and other nodes). Function `selectLV`() (Line 5 of `handleLvQS`() in Alg. 3) captures the local view selection mechanism. Please see the algorithm and implementation in §6.

MemberServ also enables ElastO nodes to form the global identifier circle. For each node $v \in V$, $2{\cdot}L$ leaf entries of $v.\mathscr{L}$ are always dedicated for maintaining the immediate neighbors on the identifier circle. Each node $v \in V$ selects L nodes with the closest nodeIds to its own, in the two directions, among the current $v.\mathscr{L}$, as its predecessors and successors. Although initially the predecessors and successors may not reflect the global circular ordering, peer sampling guarantees that the circle topology rapidly converges and is constantly maintained [14,22].

Further, MemberServ is responsible for failure detection. With the support of failure detector, each node can *locally* determine if any other node in the system is up or down. Besides, failure detector is also used to avoid attempts to communicate with unreachable nodes during various operations. Our MemberServ module can rely on standard failure detector [9]. For the rest of this paper, we assume that failure detector is available as a black box.

4.4 The Lightweight TcoUpgrader Module

Alg. 4 ElastO Local Protocol for Repairing TCO at Node $\mathbf{v} \in \mathbf{V}$

▶ $\mathbf{v}$.repairTcoOnChurn($chnevt$)
1: $c \leftarrow$ getCoordinator($chnevt$)
2: $done \leftarrow$ **false**
3: **if** $c = \mathbf{v}.self$ **then**
4: **if** readyToHandle($chnevt$) is **false then**
5: prepareLV($chnevt$)
6: **repeat**
7: $E_{new} \leftarrow$ computeTcoOnChurn($chnevt$)
8: **for all** $e(x,y) \in E_{new}$ **do**
9: send(x, **new** *AddneiQ*($\mathbf{v}.self, y$))
10: send(y, **new** *AddneiQ*($\mathbf{v}.self, x$))
11: $X \leftarrow \{x | \exists y$ s.t. $(x,y) \in E_{new}\}$
12: **wait until** [($\forall x \in X$: received *AddneiS* from x) OR ($X \wedge \mathbf{v}.\mathscr{D} \neq \emptyset$)]
13: **if** $\forall x \in X$: received matched *AddneiS* **then**
14: $done \leftarrow$ **true**
15: **until** $done$ is **true**
16: **else**
17: **repeat**
18: $c \leftarrow$ getCoordinator($chnevt$)
19: send(c, **new** *RepairS*($\mathbf{v}.self, chnevt$))
20: **wait until** [received matched *RepairS* from c OR $c \in \mathbf{v}.\mathscr{D}$]
21: **if** received *RepairS* from c **then**
22: $done \leftarrow$ **true**
23: **until** $done$ is **true**

▶ $\mathbf{v}$.getCoordinator($chnevt$)
1: **return** node descriptor $w \in (\mathbf{v}.\mathscr{L} \backslash \mathbf{v}.\mathscr{D})$ s.t. $(w.id - chnevt.node.id) > 0$ and $(w.id - chnevt.node.id)$ is minimum

▶ $\mathbf{v}$.readyToHandle($chnevt$)
1: $l \leftarrow$ get l s.t. $\mathbf{v}.pred[l] = chnevt.node$
2: **for all** $i \in \mathbf{v}.pred[l].neighbIds$ **do**
3: **if** $\nexists n \in \mathbf{v}.pneighb[l]$ s.t. $n.id = i$ **then**
4: **return false**
5: $T_B = \bigcup_{u \in \mathbf{v}.backup[l]} u.topics$
6: **if** $chnevt.node.topics \subseteq T_B$ **then**
7: **return true**
8: **else**
9: **return false**

▶ $\mathbf{v}$.prepareLV($chnevt$)
1: $w \leftarrow chnevt.node$
2: $l \leftarrow$ get l s.t. $\mathbf{v}.pred[l] = w$
3: $\mathbf{v}.pneighb[l] \leftarrow \emptyset$
4: **while** $\mathbf{v}.pneighb[l] = \emptyset$ **do**
5: $msgReq \leftarrow$ **new** *NeighbQ*($\mathbf{v}.self, w.id$)
6: send($\mathbf{v}.succ[0], msgReq$)
7: **wait until** [(received matched $msgRes$) OR ($\mathbf{v}.succ[0] \in \mathbf{v}.\mathscr{D}$)]
8: **if** received matched $msgRes$ **then**
9: $\mathbf{v}.pneighb[l] \leftarrow msgRes.neighbs$
10: **wait until** $\mathbf{v}.backup[l]$ is built, i.e., $w.topics \subseteq \bigcup_{u \in \mathbf{v}.backup[l]} u.topics$

The TcoUpgrader module is responsible for maintaining the pub/sub TCO neighborhood for each node. Upon churn, ElastO invokes repairTcoOnChurn() of the TcoUpgrader module, as shown in Alg. 4.

In repairTcoOnChurn(), node v first checks whether it is the coordinator for $chnevt$. If it is the churn coordinator, node v prepares all the knowledge required to handle this churn, and then repairs the overlay until TCO is retained – at

each attempt, it invokes TCO construction algorithm with its current local view, and then builds new overlay connections accordingly.

If v is not the churn coordinator, node v keeps trying to delegate this churn handling to another node – at each attempt, it finds a node w whose id is closet to the churn coordinator in its local view, sends a TCO repairing request message to w, and then wait a certain time for w's response.

Thanks to the nodeId circle supported by MemberServ, the churn coordinators (and thus the computation overhead of TCO maintenance and upgrade) are uniformly distributed across all nodes in ElastO. Each node is responsible for a small region on the nodeId circle – it only repairs the TCO when its predecessors join or leave. Another desirable property is that, ElastO only impacts a small subset of nodes when repairing the overlay, in which the crux of `computeTcoOnChurn()` lies. (See the algorithm design and implementation in §5.)

4.5 The TcoCreator Module

The TcoCreator module is a conceptually centralized entity to initialize (or to reset) the pub/sub TCO infrastructure at each node. It can apply any existing static algorithms or decentralized protocols (see §2).

In the implementation of TcoCreator used in this paper, we employ the LowODA algorithm [20] for initializing the base TCO from scratch. We also use LowODA as a building block in the TcoUpgrader module (see §5).

5 Overlay Repairing Algorithms in TcoUpgrader

This section concentrates on the strategies in the TcoUpgrader module to dynamically maintain the TCO in presence of churn.

Alg. 5 TCO Repairing with Shadows invoked at node $\mathbf{v}$ upon churn at $\mathbf{v}'$

▶ $\mathbf{v}$.`computeTcoOnChurn`($chnevt$)
1: $v' \leftarrow chnevt.node$
2: $l \leftarrow$ get l s.t. $v' = \mathbf{v}.pred[l]$
3: $S \leftarrow \mathbf{v}.backup[l]$
4: **if** $chnevt.type$ is JOIN **then**
5: $\quad S \leftarrow S \cup \{v'\}$
6: **else**
7: $\quad S \leftarrow S \cup \mathbf{v}.pneighb[l]$
8: $E_{cur} \leftarrow \{(x,y) | x,y \in S \wedge x.id \in y.neighbIds \wedge y.id \in x.neighbIds\}$
9: **return** `buildEdges`$(S, v'.topics, Int|_S)$

▶ $\mathbf{v}$.`buildEdges`(S, T', Int', E_{cur})
1: $E_{new} \leftarrow \emptyset$, $E_{pot} \leftarrow (S \times S) \backslash E_{cur}$
2: **while** $G = (S, E_{new})$ is not TCO **do**
3: $\quad$ **for all** $e = (x,y) \in E_{pot}$ **do**
4: $\quad\quad contrib(e) \leftarrow \big|\{t \in T' | Int'(x,t) \wedge Int'(y,t) \wedge$ x & y are in diff TC-$components$ in $G^{(t)}\}\big|$
5: $\quad e \leftarrow$ `findLowEdge`(ρ)
6: $\quad E_{new} \leftarrow E_{new} \cup \{e\}$
7: $\quad E_{pot} \leftarrow E_{pot} - \{e\}$
8: **return** E_{new}

When nodes are joining or leaving the TCO, it is possible to re-attain topic-connectivity by only adding links among a selected node subset. We try to reduce the size of the node set involved in the overlay repair algorithms. First, this allows us to repair the TCO with partial knowledge about a small subset of nodes. Second, the number of nodes turns out to be the principal factor for the time complexity of the TCO construction algorithms [6,20], and thus our algorithms can run much faster thanks to the reduced size of the node subset involved.

We define the *shadow set* as the subset of nodes that are employed for overlay repair upon node churn. For example, given an initial $TCO(V, T, Int, E)$ and an instance of churn round with only one node v', the *shadow set* (or *shadows*)

is a subset of nodes from V' that are chosen in the overlay repair step, where $V' = V \cup \{p'\}$ for joining and $V' = V \backslash \{v'\}$ for leaving. Suppose node v is the immediate successor of v' in the nodeId circle. Alg. 5 specifies the TCO repairing algorithm invoked at the coordinator node $v \in V$ for a churn event at v'. With the $v.\mathscr{L}$ maintained by MemberServ, function `computeTcoOnChurn`() first computes the *shadow set* S according to the churn type (Lines 4-7), and then repairs the TCO locally among the shadow set (Lines 8-9). In particular, Line 8 estimates the existing overlay edges among shadow set based on its partial knowledge, and function `buildEdges`() computes the edges that need to be added to restore TCO.

We implement `buildEdges`() by reusing LowODA [20]. Function `buildEdges`() operates in a greedy manner: it iteratively adds carefully selected edges one by one until topic-connectivity is attained. At each iteration, `findLowEdge`() selects an edge for the overlay (Line 5 of `buildEdges`()). The edge selection rule is based on a combination of two criteria: node degree and *edge contribution*, which is defined as reduction in the number of *TC-components* caused by the addition of the edge to the current overlay. Edge contribution for an edge e is denoted as $contrib(e)$. Function `findLowEdge`() uses a parameter ρ to tread the balance between maximum and average node degree and it makes a weighed selection between two candidate edges: (1) e_1 s.t. $contrib(e_1)$ is the maximum among E_{pot} and (2) e_2 s.t. $contrib(e_2)$ is the maximum among the subset of E_{pot} in which all edges increase the maximum degree of $G(V, E_{new} \cup E_{new})$ minimally. If $contrib(e_1) \geq \rho \cdot contrib(e_2)$, edge e_1 is added, otherwise e_2 is added.

Based on the analysis in [20], we derive that Alg. 5 efficiently achieves close-to-LowODA approximation ratios under any simple churn event (see [5]).

6 Selecting Local View in MemberServ

In order to efficiently compute the shadow set upon churn, MemberServ proactively maintains the local view $v.\mathscr{L}$ for each node $v \in V$.

Alg. 6 Selecting Local View at $\mathbf{v} \in \mathbf{V}$

```
▶ v.selectLV(buffer)
1: for l = 0 to L − 1 do
2:     v.succ[l] ← v's l-th nearest successor in (buffer\v.𝒟)
3:     v.pred[l] ← v's l-th nearest predecessor in (buffer\v.𝒟)
4:     v.backup[l] ← buildBackups(l, buffer)
5: for ∀l s.t. v.pred[l] is updated do
6:     v.pneighb[l] ← ∅, w ← v.pred[l]
7:     msgReq ← new NeighbQ(w.id)
8:     send(w, msgReq)
9:     wait until [(received expected msgRes from w) OR (w ∈ v.𝒟)]
10:    if received expected msgRes then
11:        v.pneighb[l] ← msgRes.neighbs
```

We can regard the shadow set S (Line 5 and 7 in Alg. 5) as a sample of nodes that is representative of specific characteristics for the entire node population V'. Taking topic-connectivity into account, it is always safe (but not necessarily efficient) to set shadow set as the complete node set V'. However, there may exist many other choices for the shadow set with much fewer nodes. In the case that node v' is leaving, one candidate for the shadow set is the *neighbor set* around the leaving node v', denoted as $v'.\mathscr{N}$. Observe that to re-attain topic-connectivity, it is sufficient to add links among $v'.\mathscr{N}$ to the existing overlay. This

can be done very efficiently since $v'.\mathcal{N}$ is usually much smaller than the complete node set V', but the node degrees of $v'.\mathcal{N}$ would usually degrade significantly in the output TCO. The trade-off between the runtime cost and the quality of the output TCO can be balanced by selecting the shadow set in between $v'.\mathcal{N}$ and V'. In the example of Fig. 1, when Node v_{10} is leaving, the neighbor set $v_{10}.\mathcal{N} = \{v_{11}, v_3, v_5\}$, in conjunction with a subset of other nodes $B = \{v_1, v_6\}$, forms the shadow set S to repair the broken TCO.

Function `buildBackups`() in Alg. 6 builds the backup set for each predecessor. For each $v' = v.pred[l]$ where $0 \leq l < L$, we want the backup set $B(v') = v.backup[l]$ to possess two desirable properties: 1. Each $u \in B(v')$ shares at least one topic with v': $(u.topics \wedge v'.topics) \neq \emptyset$, $\forall u \in B(v')$; 2. All topics subscribed by v' are covered by $B(v')$: $v'.topics \subseteq (\cup_{u \in B(v')} u.topics)$.

Note that B with the above two properties is equivalent to a *set cover* where v''s topics is the universe of ground elements that are supposed to be covered by the topic set of backup nodes. The *minimum weighted set cover* problem is a well-studied NP-hard problem [8,12]. We can therefore apply classical algorithms to obtain a *feasible* (not necessarily minimum) backup set.

Still, there is a trade-off concerning the size of the backup set. On the one hand, small cardinality is desired for the backup set with regard to runtime cost, because the number of backups directly impacts the size of the shadow set and thus the time complexity of the overlay repair algorithms. On the other hand, a sufficiently large backup set is preferred for ensuring the quality of the output TCO: a larger backup set means a large shadow set S and therefore a higher probability for the instance $(S, v'.topics, Int|_S)$ to approximate $I'(V', T, Int')$ with regard to the maximum and average node degrees in the TCO.

We introduce the *coverage factor* to tune the size of the backup set for a node and seeking balance between the time complexity of the overlay maintenance and the quality of the output TCO. Given $I(V, T, Int)$, we build the backup set $B(v')$ for each node $v' \in V$. The coverage factor for the backup set $B(v')$, denoted as $\lambda(B(v'))$ (or λ), is the minimum number of subscribers to t within $B(v')$ taken across all $t \in v'.topics$: $\lambda(B(v')) = \min_{t \in v'.topics} |\{u | u \in B(v') \wedge t \in u.topics\}|$.

The *coverage factor* is an integer ($\lambda \geq 0$) such that each topic of interest of v' is covered at least λ times by its backup set $B(v')$.

The coverage factor selection exhibits the trade-off between the running time and node degrees: On the one hand, $\lambda = 0$ minimizes the size of the backup set and running time, but leads to a severe impact on the node degrees. On the other hand, if we choose the coverage factor to be a large value such that all nodes of V' have to be included in the backup set, then both the maximum and average node degrees are close to those generated by running the static algorithms from scratch, but the runtime cost is not insignificant. According to our experiments in §7.3, an increase in λ beyond 5 only marginally improves the node degrees of the TCO under churn. The backup set for $\lambda = 5$ is significantly smaller than the complete node set itself so that we choose 5 as the default value for λ.

There exist two efficient approximation algorithms for the *minimum weight set cover* problem: the *greedy* algorithm [8] and the *primal-dual* algorithm [12]. Function `buildBackups`() has two implementations - each executes the greedy or

primal-dual algorithm iteratively until a λ-*backup-set* is obtained, which we refer to as `buildBackupsGreedily`() and `buildBackupsByPD`(), respectively.

The `buildBackupsGreedily`() procedure applies the *greedy* set cover heuristic for constructing a backup set [8]. It always chooses the next node w so as to provide the minimum average weight over uncovered topics that would be covered by w. This algorithm achieves a log approximation ratio. However, its greediness leads to prioritizing bulk nodes that subscribe to a large number of topics upon backup selection. A small number of bulk nodes would serve as backups for a large number of nodes while the majority of lightweight nodes would not be selected as backups at all. Fairness is lost to a large extent in this case. Furthermore, the impact of accumulated sub-optimality would become progressively severe over time as the number of churn rounds increases.

The `buildBackupsByPD`() procedure is based on the *primal-dual* method for computing minimum weight set cover [12]. The algorithm proceeds in an iterative manner: each time randomly picking an uncovered topic t and choosing a subscriber w for t with the minimum weight as a backup. The primal-dual algorithm yields an f-approximate solution for minimum weight set cover where f is the maximum frequency of any element. The approximation ratio of primal-dual is higher than that of the greedy set cover algorithm. Yet, it is deemed acceptable in practice for many instances of the problem. Moreover, the primal-dual approach integrates randomness into greediness and effectively mitigates the prioritization of bulk subscribers. Therefore, we decide to leverage the primal-dual algorithm towards building the backup set in Line 4 of Alg. 6. We experimentally compare these two algorithms in §7.2.

7 Evaluation

Table 2: Algorithms and protocols that we evaluate

ElastO	**The pub/sub TCO maintenance system proposed in this work**
ElastO-L	Local view is maintained at each node for computing shadows and TCOs.
ElastO-G	Global view is maintained at each node for computing shadows and TCOs.
LowODA [20]	**Low Maximum and Average Degree Overlay Design Algorithm**
LowODA-Inc	Incrementally repair TCO using LowODA-rule regarding existing edges.
LowODA-Re	Reconstruct TCO from scratch at each churn round regardless of existing edges.
SpiderCast [7]	**A peer-to-peer pub/sub overlay construction protocol**
SpiderCast(K_g, K_r)	Neighbor selection combines two *local* heuristics: *greedy* and *random coverage*. Each node tries to cover its interested topics K_g or K_r times.

We implement ElastO and other comparison algorithms and protocols in Java (see Table 2). We simulate large-scale peer-to-peer networks with PeerSim [21]. We use LowODA [20] as a baseline because it is the only known polynomial time algorithm that achieves sub-linear approximation ratios on both the maximum and average node degrees. We implement two LowODA based algorithms for handling churn: LowODA-Inc and LowODA-Re. We also compare the performance of ElastO with SpiderCast [7] because it is highly efficient in maintaining TCO in a decentralized peer-to-peer manner and has been adopted in practice [11].

We evaluate the above systems with both synthetic pub/sub workloads and real-world traces derived from data sets of Facebook, Twitter and Google. We

mainly focus on evaluating the overlay properties (e.g., the node degree and the diameter) and the efficiency of repairing topic-connectivity under churn.

7.1 Experimental Setup

Pub/Sub Workloads: To represent typical pub/sub workloads, we synthetically generate three types of topic popularity distributions: uniform, Zipfian, and exponential. We also extract the workloads from real-world social networks, namely Twitter and Facebook.

(1) *Synthetic workloads:* We initialize the base instance $I_0(V_0, T, Int)$ with $|V_0|$=2 000, $|T|$=200 and $|v.topics| \in [10, 90]$, where the subscription size of each node follows a power low. Each topic $t \in T$ is associated with probability $p(t)$, $\sum_{\in T} p(t)$=1, so that each node subscribes to t with a probability $p(t)$. The value of $p(t)$ is distributed according to either an exponential, a Zipfian (with α=2.0), or a uniform distribution, which we call Expo, Zipf, or Unif for short. According to [7], these distributions are representative of actual workloads used in industrial pub/sub systems today. Expo is used by stock-market monitoring engines for the study of stock popularity in the New York Stock Exchange [26]. Zipf faithfully describes the feed popularity distribution in RSS feeds [19].

(2) *Facebook dataset*: We use a public Facebook dataset [27], with over 3 million distinct user profiles and 28.3 million social relations as a second workload for our evaluations. In Facebook, when a user has some activity (e.g., updating the status, sharing new photos, or commenting on a blog), all her friends will receive some notifications as subscribed. As such, we model each user, say Alice, as a topic, and all her friends are the respective subscribers. Likewise, the friend set of Alice forms her subscription set. The Facebook relations are bidirectional, so friends in the Facebook social graph subscribe to each other in our model.

(3) *Twitter dataset*: We also use a public Twitter dataset [15], containing 41.7 million distinct user profiles and 1.47 billion social followee/follower relations. Similarly to Facebook, users are modeled as topics as well as subscribers. However, in Twitter relations are unidirectional, i.e., user Alice following user Bob does not require that Bob also follows Alice.

We extract the workloads from the original Facebook and Twitter social graphs with a methodology inspired from [22]. More specifically, starting with a random set of a few users as seeds, we traverse the social graph via breadth first search, until it reaches the target number of nodes, and our sample includes all edges among the nodes. The size of our samples is 1K or 10K, i.e., $|V| \approx$ 1K and $|T| \approx$ 1K, OR $|V| \approx$ 10K and $|T| \approx$ 10K. Fig. 2 shows the complementary cumulative distribution function (CCDF) of follower/followee counts for both the original Facebook and Twitter datasets, and for our extracted datasets in the inner plot. The plots indicate that the original dataset properties were retained in the extracted sample. We also observe from Fig. 2 that the Twitter data

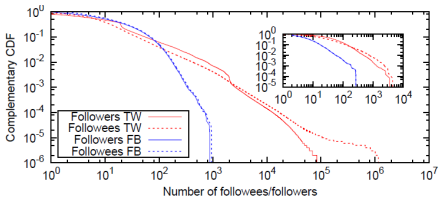


Fig. 2: CCDF of followers and followees: Twitter (41.7M users) and Facebook (3M users). Inner plot: 10K-user sample.

has more correlation than the Facebook data. We denote the instances of our samples by FB 1K, FB 10K, TW 1K, and TW 10K.

Churn traces: We first generate sample churn traces that contain over 1 000 rounds of churn. Each churn round generates a join or leave with probability 0.5.

We also use the Google cluster data [1] as real-world churn traces to evaluate our system under churn. The Google cluster trace includes data from an 11K-machine cell over about a month-long period. The machine event table contains three types of machine events:

– ADD: a machine became available in the cluster.
– REMOVE: a machine was removed from the cluster.
– UPDATE: a machine available in the cluster had its available resources changed, such as CPU, memory, disk space, etc.

We randomly sampled 1K and 10K unique machine ids from the start of the Google cluster trace and then generated join churn rounds via ADD events and leave churn rounds via REMOVE events. We omitted UPDATE events.

7.2 Building Backups Greedily vs. by Primal-dual

We first compare two algorithms for building backups as we propose in §6. To eliminate other factors in the design domain, we deploy two instances of ElastO-G with `buildBackupsGreedily()` and `buildBackupsByPD()`, respectively. For both instances, we set λ=3, initiate the same TCOs, and feed identical churn traces.

As a complement measure to the backup set, we define the *primary set* for each node $v \in V$ as a subset of nodes for which v serves as backup in the local view, i.e., $P(v) = \{u | v \in B(u)\}$. We call $|P(v)|$ as v's primary degree.

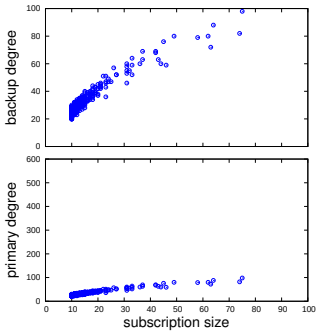


Fig. 3: `buildBackupsByPD()`

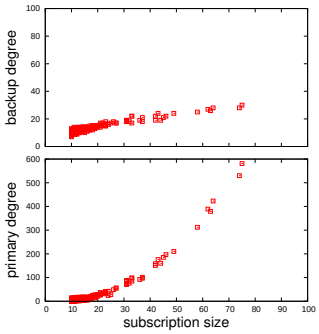


Fig. 4: `buildBackupsGreedily()`

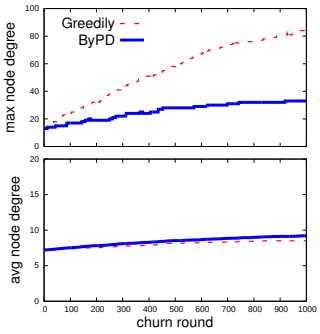
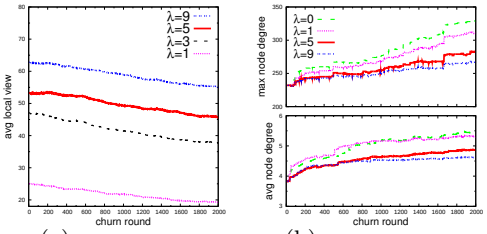


Fig. 5: Node degrees

Fig. 3 and Fig. 4 show the output of primary-backup schemes under Unif, respectively. As shown in the figures, both `buildBackupsGreedily()` and `buildBackupsByPD()` produce small-sized backup sets compared to the complete node set V_0: 2.8% of $|V_0|$ for `buildBackupsByPD()` and 1.1% of $|V_0|$ for `buildBackupsGreedily()` on average across all nodes, respectively. In general, the backup degree is linearly proportional to the subscription size for both algorithms. However, the distribution of the primaries differs considerably between them. The primaries produced by `buildBackupsGreedily()` are skewed and follow an exponential shape: 42.9% of all nodes have less than 5 primaries, while the highest primary degree of a single bulk subscriber peaks at 581. At the same time, the distribution of the

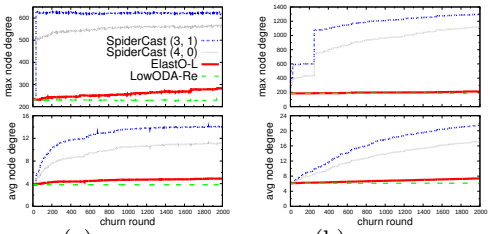
backups produced by `buildBackupsByPD`() is well-balanced: the lowest primary degree is 21 and the highest is 59. These results confirm our observations in §6 about the side effect of greediness on the primary assignment and the fairness introduced by randomness in the primal-dual scheme.

We also compare both backup construction algorithms in terms of the evolution of overlay properties under churn. As shown in Fig. 5, both the maximum and average node degrees increase as the TCO instance evolves with node churn. However, overlay quality for ElastO-G utilizing `buildBackupsGreedily`() degrades noticeably, i.e., at churn round 1000, the maximum node degree becomes 84. On the other hand, when the backups are built by primal-dual, both the maximum and average degrees keep a steadily low growth rate. Results presented in Fig. 3, 4 and 5 substantiate our choice of `buildBackupsByPD`() over `buildBackupsGreedily`(). In the rest of the evaluation, we choose `buildBackupsByPD`() in the `selectLV`() procedure (Line 4 of Alg. 6).



(a) Local view (b) Node degrees

Fig. 6: ElastO-L with λ's - FB 1K



(a) FB 1K (b) FB 10K

Fig. 7: Node degrees under churn

7.3 Selecting Local View and Choosing Coverage Factor

We explore the impact of local view selection with different coverage factors on the output and performance of ElastO-L. We evaluate ElastO-L with different values of the coverage factor ($\lambda = 0, ..., 9$).

Fig. 6(a) shows that the average local view maintained in ElastO-L is fairly small as compared to the overall size of the network. Under FB 1K, the average local view is 21.02 with $\lambda = 1$, 40.49 with $\lambda = 3$, 48.40 with $\lambda = 5$, and 57.59 with $\lambda = 9$, on average. The average local view also decreases over time, and the majority of nodes keeps a small local view steadily under all instances. This is because, ElastO-L evolves with constant epidemic exchange, and the local view converges promptly with more balanced load distribution. Under all instances, after the first 100 rounds of gossiping, more than 95% of the nodes in ElastO-L$|_{\lambda=5}$ have their local view fewer than 5% of all nodes. This demonstrates that ElastO-L achieves good load balancing in terms of local view maintenance.

Fig. 6(b) shows that, as λ increases, $D_{\text{ElastO-L}}$ decreases, and its growth rate with respect to the churn round decreases. The differences in the maximum node degrees and their growth rates also decrease with successive coverage factors. Under FB 1K, at the end of the Google cluster churn trace, $D_{\text{ElastO-L}}|_{\lambda=0} = 333$, $D_{\text{ElastO-L}}|_{\lambda=1} = 313$, $D_{\text{ElastO-L}}|_{\lambda=3} = 302$, $D_{\text{ElastO-L}}|_{\lambda=5} = 283$, $D_{\text{ElastO-L}}|_{\lambda=7} = 277$, and $D_{\text{ElastO-L}}|_{\lambda=9} = 274$. When $\lambda \geq 5$, the difference of the maximum node degrees with different λ values is insignificant. The average node degrees follow

the same trends with respect to the coverage factor, and the difference among different values of λ is even more insignificant.

These experiments demonstrate our `selectLV()` method achieves good load balancing in terms of local view maintenance across all nodes in ElastO, good balancing of , and confirms the validity of choosing a relatively small coverage factor (see §6). We fix $\lambda = 5$ for ElastO-L in the rest of §7, which demonstrates the scalability, efficiency and robustness of ElastO-L in presence of churn.

7.4 Overlay Node Degrees

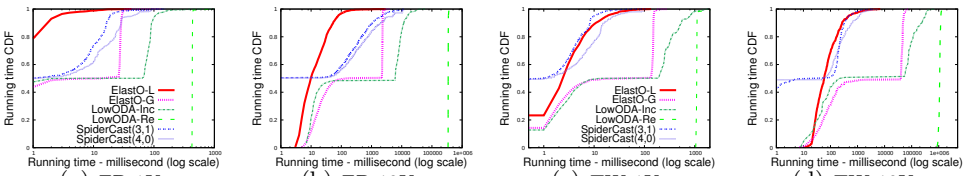
Fig. 7 compares the node degrees produced by different algorithms and protocols as the instances evolve with the Google churn trace, where we initialize all with the same TCO constructed by LowODA from scratch. We do not plot lines for ElastO-G and LowODA-Inc, because the distances from ElastO-G to ElastO-L (or from LowODA-Inc to LowODA-Re) are too small to distinguish, e.g., $D_{\text{ElastO-L}} \leq 1.04 \cdot D_{\text{ElastO-G}}$, and $\overline{d}_{\text{ElastO-L}} - \overline{d}_{\text{ElastO-G}} \leq 0.76$, on average under FB 1K .

First, ElastO-L output similar maximum and average node degrees as compared to LowODA-Re. For example, $D_{\text{ElastO-L}} \leq 1.15 \cdot D_{\text{LowODA-Re}}$, $\overline{d}_{\text{ElastO-L}} - \overline{d}_{\text{LowODA-Re}} \leq 1.12$, on average over the entire churn sequence under FB.

Second, the node degrees of ElastO-L degrade *slowly* like a step function along the churn rounds. From churn round 1 to 2000 under FB 1K, $D_{\text{ElastO-L}}$ increases from 232 to 283, a degradation rate of 0.025 per churn round. This rate becomes even *slower* as the input instance scales up, which drops to 0.013 under FB 10K.

Third, SpiderCastdegrades fast in the first 200 churn rounds, and the output node degrees stay a number of times higher than those of ElastO and LowODA. The gap increases as the instances scale up. For example, $D_{\text{SpiderCast}(3,1)} - D_{\text{ElastO-L}}$ is 355 under FB 1K and 1050 under FB 10K, respectively on average. SpiderCast$(3, 1)$ generates more edges than SpiderCast$(4, 0)$ due to random coverage, which leads to a better chance to attain topic-connectivity (see [7]).

7.5 Runtime Cost



(a) FB 1K (b) FB 10K (c) TW 1K (d) TW 10K

Fig. 8: Runtime cost

Fig. 8 depicts *cumulative distribution functions* (CDF) for the running time of different algorithms over a sequence of churn rounds. As shown in Fig. 8, LowODA-Re runs considerably slower than other dynamic algorithms, because LowODA-Re tears down existing links and reconstructs the TCO from scratch at each churn round. The runtime costs of LowODA-Inc and ElastO-G are of the same order of magnitude. ElastO-L and SpiderCast further improve the time efficiency significantly – ElastO-L is around 4.67% that of ElastO-G on average across all instances, thanks to local operations rather than global computation.

The speedup of ElastO-L as compared to the static LowODA-Re is more profound when the size of instance increases from 1K up to 10K. The running time ratio of ElastO-L against LowODA-Re is around 0.20% under 1K and 0.01% under 10K, on average. This demonstrates the scalability of ElastO-L with respect to the number of nodes, the number of topics, and the subscription sizes.

ElastO, SpiderCast and LowODA-Inc require more time to dynamically repair the TCO under leaves than under joins. This can be explained by the number of nodes involved in the repairing phase. Different magnitudes of time costs between handling joins and leaves form clear horizontal lines around 50% in the CDFs of ElastO-G, SpiderCast, and LowODA-Inc. Meanwhile, the shape of running time CDF for ElastO-L exhibits more smoothness and robustness across all churn rounds. This shows that ElastO-L achieves balanced load distribution and fairness among all nodes in the network over the sequence of churns.

7.6 Topic Diameters

We also look at *topic diameters*, which impact many performance factors for efficient routing in pub/sub, e.g., message latency. Given $TCO(V, T, Int, E)$, the topic diameter for $t \in T$ is $diam^{(t)} = diam(G^{(t)})$, where $diam(G^{(t)})$ is the maximum shortest distance between any two nodes in $G^{(t)}$. We denote the maximum and average topic diameter across all topics as $Diam$ and $\overline{diam}$, respectively.

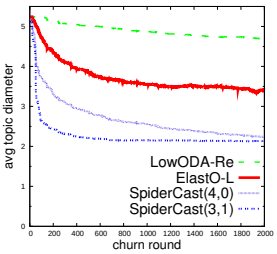


Fig. 9: Diameter - TW 1K

Fig. 9 depicts the average topic diameters produced by different algorithms and protocols over the Google churn sequence under TW 1K. All systems start with the same topic diameters, which tend to to decrease as the overlays evolve with the churn, because the number of overlay edges increases. Generally, topic diameters are inversely proportional to node degrees: SpiderCast has the lowest topic diameters, and LowODA-Re has the highest topic diameters. Topic diameters of ElastO-L are slightly higher than those of SpiderCast, but the difference is insignificant: $Diam_{\text{ElastO-L}} - Diam_{\text{SpiderCast}(3,1)} = 3.67$, and $\overline{diam}_{\text{ElastO-L}} - \overline{diam}_{\text{SpiderCast}(3,1)} = 1.45$, respectively on average.

8 Conclusions

We present a fully *dynamic* system, ElastO, to construct and maintain low fanout TCOs for pub/sub systems under churn. We demonstrate that the ElastO system combines the advantages from both the static algorithms and the decentralized protocols under large-scale pub/sub workloads extracted from Twitter and Facebook and real-world cluster churn traces released by Google: (a) both the maximum and average node degrees remain insignificantly higher than those of the *static* algorithms; (b) the time efficiency is of the same order of magnitude as the decentralized protocols.

References

1. Google Cluster Data. http://code.google.com/p/googleclusterdata/.

2. M. Allani, B. Garbinato, and P. Pietzuch. Chams: Churn-aware overlay construction for media streaming. *Peer-to-Peer Networking and Applications*, 2012.
3. E. Baehni, P. Eugster, and R. Guerraoui. Data-aware multicast. In *DSN'04*.
4. M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *JSAC*, 2002.
5. C. Chen and R. Vitenberg. Dynamic maintenance of topic-connected overlays under churn. Technical report, 2013. http://msrg.org/papers/TRCJV-Shadow.
6. G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Constructing scalable overlays for pub-sub with many topics: Problems, algorithms, and evaluation. In *PODC'07*.
7. G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication. In *DEBS'07*.
8. V. Chvtal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 1979.
9. X. Défago, P. Urbán, N. Hayashibara, and T. Katayama. Definition and specification of accrual failure detectors. In *DSN*, pages 206–215, 2005.
10. S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP'03*.
11. G. P. Guido Urdaneta and M. van Steen. Towards a fully decentralized and collaborative hosting infrastructure for Wikipedia. In *WikiSym'08*.
12. D. S. Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on Computing*, 1982.
13. M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware '04*.
14. M. Jelasity, A. Montresor, and O. Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer Networks*, pages 2321–2339, 2009.
15. H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10*.
16. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating System Review*, 2010.
17. J. Leitao, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *DSN '07*.
18. D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *PODC*, 2002.
19. H. Liu, V. Ramasubramanian, and E. G. Sirer. Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In *IMC'05*.
20. M. Onus and A. W. Richa. Parameterized maximum and average degree approximation in topic-based publish-subscribe overlay network design. In *ICDCS'10*.
21. A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *P2P'09*.
22. F. Rahimian, S. Girdzijauskas, A. H. Payberah, and S. Haridi. Vitis: A gossip-based hybrid overlay for internet-scale publish/subscribe enabling rendezvous routing in unstructured overlay networks. In *IPDPS '11*.
23. J. Reumann. Pub/Sub at Google. Lecture & Personal Communications at EuroSys & CANOE Summer School, Oslo, Norway, Aug 2009.
24. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware '01*.
25. V. Setty, M. van Steen, R. Vitenberg, and S. Voulgaris. Poldercast: Fast, robus, and scalable architecture for p2p topic-based pub/sub. In *Middleware'12*.
26. Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS*, 2005.
27. C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EuroSys '09*.