Churn Possibilities and Impossibilities*

Dianne Foreback[1], Mikhail Nesterenko[1], and Sébastien Tixeuil[2]

[1] Kent State University, Kent, OH, USA,
mikhail@cs.kent.edu,
[2] Sorbonne Université, CNRS, LIP6, FR-75005, France

Abstract. Churn is processes joining or leaving the peer-to-peer overlay network. We study handling of various churn variants. Cooperative churn requires leaving processes to participate in the churn algorithm while adversarial churn allows the processes to just quit. Infinite churn considers unbounded number of churning processes throughout a single computation. Unlimited churn does not place a bound on the number of concurrently churning processes. Fair churn handling requires that each churn request is eventually satisfied. A local solution involves only a limited part of the network in handing a churn request.

We prove that it is impossible to handle adversarial unlimited churn. We sketch a global solution to all variants of cooperative churn and focus on local churn handling. We prove that a local fair solution to infinite churn, whether limited or unlimited, is impossible. On the constructive side, we present an algorithm that maintains a linear topology and handles the least restrictive unfair churn: infinite and unlimited. We extend this solution to a 1-2 skip list, describe enhancements for generalized skip lists and skip graphs.

1 Introduction

In a peer-to-peer overlay network, each member maintains the identifiers of its overlay neighbors in its memory while leaving message routing to the underlay. Such a network is inherently decentralized and scales well. Peer-to-peer overlays are well suited for distributed content storage and delivery. Their recent applications range from internet telephony [8] to digital cryptocurrencies [29]. Due to the lack of central authority and the volunteer nature of overlay network participation, *churn*, or joining and leaving of peers, is a particularly vexing problem. Churn may be cooperative, if departing processes execute a prescribed departure algorithm; or adversarial, if they just quit.

Infinite and unlimited churn. Every peer-to-peer overlay network has to handle churn. Usually, while the topological changes in the overlay required by the churn request occur, the primary services of the overlay, such as content retrieval, are either suspended or disregarded altogether. In other words, the churn is considered finite and the overlay network users have to wait till join/leave requests stop coming. Then, the overlay network recovers and restores services. This may

* Preliminary fragments of this work appeared in [17].

be tolerable if churn is infrequent since the overlay network is available most of the time. However, at the scales that peer-to-peer overlay networks achieve, churn is frequent if not continuous. In this case, churn related service degradation may become unacceptable. It is, therefore, necessary to consider *infinite churn* under which the overlay network has to maintain services while handling it.

One way to handle churn is to engineer sufficient redundancy in the overlay network topology so that if peers leave or join, there are enough alternative paths for the operation of the network to proceed uninterrupted. In this approach, the amount of redundancy necessarily places a limit on the number of processes that churn concurrently: the churning processes must not sever all redundant paths. If this limit is breached due to extensive churn, the network may collapse and partition itself. To prevent such an outcome, the redundancy has to be extensive. However, in the absence of heavy churn, this redundancy wastes resources. In this paper, we consider *unlimited churn* with no bound on the number of concurrently joining or leaving processes.

Unfair and local churn. In cooperative churn, the joining or leaving peer submits a request to the churn handling algorithm. Such an algorithm is *fair* if it eventually satisfies every such request. A fair algorithm may not always be possible or efficient. An *unfair* churn handling algorithm may guarantee progress by satisfying some requests but denying others indefinitely. A *global* churn handling algorithm may designate a single process to handle all churn requests. Although such a serial request handling solution may be simple, it may not be practical as it creates a performance bottleneck and a single point of failure. In contrast, a *local* solution only involves processes in the vicinity of the churning process. In this paper, we study fairness and locality of churn solutions.

Topologies. An ad hoc peer-to-peer network forms haphazardly. A structured peer-to-peer network maintains a particular topology to optimize its performance. Most structured networks start with peer linearization [19] and then add skip-links for search acceleration [1, 5, 34, 37]. A skip-list and skip-graph [13, 24, 30] are examples of a structured network built in this manner. Handling churn in a skip-list extends to other similarly built structured networks in a straightforward manner.

<table>
<tr><th colspan="2"></th><th>finite
limited or unlimited</th><th>infinite
limited or unlimited</th></tr>
<tr><td colspan="2">global</td><td colspan="2">possible, Proposition 1</td></tr>
<tr><td rowspan="2">local</td><td>unfair</td><td rowspan="2">possible, Theorems 3 and 4</td><td></td></tr>
<tr><td>fair</td><td>impossible,Theorem 2</td></tr>
</table>

Fig. 1. Cooperative churn solutions summary.

Our contribution. We consider the problem of churn in structured peer-to-peer overlay networks in the asynchronous message passing system model. We

first prove that there does not exist an algorithm that can handle unlimited adversarial churn. We then focus on cooperative unlimited churn. Our results are summarized in Figure 1. We outline the solution to global unlimited churn and focus on local solutions. We distinguish fair and unfair types of the problem. We prove that there is no local solution to the Fair Infinite Churn Problem regardless of whether it is limited or unlimited. We then present an algorithm that solves the unfair version of the problem while maintaining a linear topology, i.e. topological sort. This solution immediately applies to fair and finite churn. We extend our algorithm to handle churn in a more efficient structure of a 1-2 skip list. We describe solutions for generalized skip lists and skip graphs.

To the best of our knowledge, this paper is one of the first to focus specifically on churn and is the first systematic study of unlimited infinite churn.

Related work. Independently of peer-to-peer overlay networks, several papers [25, 28, 38] address determination of the rate of churn, which is a difficult task itself. Churn is studied for some fundamental problems in distributed computing such as Agreement [3, 4, 21]. Churn can potentially be addressed by the solution to the Group Membership Problem [11] or an implementation of a perfect failure detector [12]. However, the studied problems are inherently global, which makes them unsuitable for peer-to-peer network use.

Peer-to-peer overlay networks are often designed to have redundant links so that they can withstand limited churn [5–7, 20]. Many papers address repairing the topology after determining a process unexpectedly left the overlay network [1, 4, 15, 22, 34, 35]. Others limit churn to maintain overlay services while adjusting the network [2, 27].

An alternative approach is to self-stabilize from churn. Self-stabilization allows the peer-to-peer network to recover from an arbitrary state once the disruptions cease [9, 10, 14, 16, 19, 23, 24, 26, 30, 31, 33, 36]. Using oracles allows a peer-to-peer network to recover from an initial incorrect state, even disconnection [16, 31]. A general framework of dealing with node departures is discussed [9, 26]. These approaches address finite churn.

Thus, previously, studies focused on limited or finite churn, while this paper focuses on unlimited and infinite churn.

2 Model and Problem Statement

Peer-to-peer overlay networks, topology. A peer-to-peer overlay network consists of a set of processes with unique identifiers. When it is clear from the context, we refer to processes and their identifiers interchangeably. Processes communicate by message passing. A process stores identifiers of other processes in its memory. Process a is a *neighbor* of process b if b stores the identifier of a. Note that b is not necessarily a neighbor of a. A process may send a message to any of its neighbors. Message routing from the sender to the receiver is carried out by the underlying network. A process may send a message only to the receiver with a specific id, i.e. we do not consider broadcasts or multicasts. Communication channels are FIFO with unlimited message capacity. A *structured* peer-to-peer overlay network maintains a particular topology. One of the

basic topologies is *linear*, or a topological sort, where each process b has two neighbors $a < b$ and $b < c$ such that a is the highest id in the overlay network that is less than b and c is the lowest id greater than b.

Consider a particular topology. A *cut-set* is a (proper) subset of processes of the network such that the removal of these processes and their incident edges disconnects the network. It is known that if a network topology is not a complete graph, it has a cut-set. Since a peer-to-peer overlay network maintains its connectivity by storing identifiers in the memory of other processes, once disconnected it may not re-connect. Hence, a peer-to-peer overlay network must not become disconnected either through the actions of the algorithm or through churn actions.

Searching, joining and leaving the overlay network. The primary use of a peer-to-peer overlay network is to determine whether a certain identifier is present in the network. A search request message bearing the identifier of interest may appear in the incoming channel of any process that has already joined the overlay network. The request is routed until either the identifier is found or its absence is determined.

A process may request to join the overlay network. We abstract bootstrapping by assuming that a join request, bearing the joining process identifier, appears in an incoming channel of any process that has already joined the overlay network. A process that joined the overlay network may leave it in two ways. In *adversarial churn* a leaving process just exits the overlay network without participating in further algorithm actions. In *cooperative churn* a leaving process sends a request to leave the overlay network; the leaving process exits only after it is allowed to do so by the algorithm. A process may join the overlay network and then leave. However, a process that left the overlay network may not join it again with the same identifier. A join or leave request is a *churn request* and the corresponding join or leave message is a *churn message*. When a leaving process exits the overlay network, the messages in its incoming channels are lost. However, the messages sent from this process before exiting remain in the incoming channel of the receiving process.

Churn algorithm. A churn algorithm handles churn requests in cooperative churn. For each process, an algorithm specifies a set of variables and actions. An *action* is of the form $\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$ where *label* differentiates actions, *guard* is a predicate over local variables, and *command* is a sequence of statements that are executed *atomically*. The execution of an action transitions the overlay network from one state to another. An algorithm *computation* is an infinite fair sequence of such states. We assume two kinds of fairness of computation: weak fairness of action execution and fair message receipt. *Weak fairness* of action execution means that if an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. *Fair message receipt* means that if the computation contains a state where there is a message in a channel, this computation also contains a later state where this message is not present in the channel, i.e. there is no message loss and the mes-

sage is received. We place no bounds on message propagation delay or relative process execution speeds, i.e. we consider fully asynchronous computations.

Algorithm locality. A churn request may potentially be far, i.e. a large number of hops, from the place where the topology maintenance operation needs to occur. *Place of join* for a join request of process x, is the pair of processes y and z that already joined the overlay network, such that y has the greatest identifier less than x and z has the smallest identifier greater than x. In every particular state of the overlay network, for any join request, there is a unique place of join. Note that as the algorithm progresses and other processes join or leave the overlay network, the place of join may change. *Place of leave* for a leave request of process x is defined similarly. *Place of churn* is a place of join or leave.

A network topology is *expansive* if there exists a constant m independent of the network size such that for every pair of processes x and y where the distance between x and y is greater than m, a finite number of processes may be added m hops away from x and the same number of processes may be removed from the network such the distance between x and y is increased by at least one. This constant m is the *expansion vicinity* of the topology. In other words, in an expansive topology, every pair of processes far enough away may be further separated by adding processes to the network while removing processes elsewhere. Note that a completely connected topology is not expansive since the distance between any pair of processes is always one. However, a lot of practical peer-to-peer overlay network topologies are expansive. For example, a linear topology is expansive with expansion vicinity of 1 since the distance between any pair of processes at least two hops away may be increased by one if a process is added outside the neighborhood of one member of the pair.

A churn algorithm is *local* if there exists a constant l independent of the overlay network size, such that only processes within l hops from the place of churn need to take steps to satisfy this churn request. The maximum such constant l is the *locality* of the algorithm. Note that a local algorithm may maintain only an expansive topology, and that the expansive vicinity of this topology must not be greater than the locality of the algorithm.

Orthogonality of infinite and unlimited churn. A churn algorithm is designed to handle particular churn. Churn is *infinite* if the number of churn requests in a computation is not bounded by a constant either known or unknown to the algorithm. To prevent the degenerate case of an indefinitely expanding network, we assume that the difference between the number of join and leave requests is still bounded. Churn is *unlimited* if the number of concurrent churn requests in the overlay network is not bounded by a constant either known or unknown to the algorithm. Observe that unlimited churn allows, for example, that all process of the network request to leave. For limited churn, we assume that there is a number $k > 1$ such that in any computation, the number of concurrent requests is no more than k.

Note that these pairs of conditions are orthogonal. For example, churn may be finite but unlimited: all processes may request to leave but no more join or leave requests are forthcoming. Alternatively, in infinite limited churn, there may

be an infinite total number of join or leave requests but only, for example, five of them in any given state.

The problem statements. A *link* is the state of channels between a pair of neighbor processes. As a churn algorithm services requests, it may temporarily violate the overlay network topology that is being maintained. A *transitional link* violates the overlay network topology while a *stable link* conforms to it. An algorithm that solves a particular churn problem conforms to the following properties.

request progress: if there is a churn request in the overlay network, some churn request is eventually satisfied;
fair request: if there is a churn request in the overlay network, this churn request is eventually satisfied;
terminating transition: every transitional link eventually becomes stable;
message progress: a message in a stable link is either delivered or forwarded closer to the destination;
message safety: a message in a transitional link is not lost.

Note that the fair request property implies the request progress property. The converse is not necessarily true. The following combinations of properties are of particular interest.

Definition 1. *A solution to the* Unfair Churn Problem *satisfies the combination of request progress, terminating transition, message progress and message safety properties.*

Definition 2. *A solution to the* Fair Churn Problem *satisfies the combination of fair request, terminating transition, message progress and message safety properties.*

In other words, a solution to the Fair Churn Problem guarantees that every churn request is eventually satisfied while a solution to the Unfair Churn Problem does not. An algorithm may satisfy these properties while handling finite or infinite, limited or unlimited churn. Note that if a solution is proven impossible under more restrictive churn conditions, it is also impossible under less restrictive conditions. For example, if the solution to the Fair Churn Problem cannot handle limited churn, it cannot handle unlimited churn either. Conversely, if a solution is proven to handle less restrictive conditions, it is guaranteed to handle more restrictive conditions. For example, if the solution to the Unfair Churn Problem handles infinite unlimited churn, it also handles limited and finite churn.

3 Impossibilities and Global Solutions

Adversarial churn.

Theorem 1. *There does not exist a solution for unlimited adversarial churn if the maintained topology is not fully connected.*

Formal proofs are in the full version of the paper [18]. Intuitively, the reason for the negative result of Theorem 1 is as follows. So long as the network is not completely connected, there is a subset of nodes whose abrupt departure may disconnect the network. For the rest of the paper, we are focusing on cooperative churn.

Theorem 2. *There does not exist a local solution to the Fair Churn Problem that can handle infinite limited or unlimited churn for an expansive overlay network topology.*

The intuition for Theorem 2 is that, in an expansive overlay network topology, the requests may arrive to produce a "treadmill effect" for a particular churn request r: the satisfaction of inopportune requests by a local algorithm extends the topology such that r never reaches its place of churn. Hence, no fairness.

Global churn handling.

Proposition 1. *There exists a global Fair Churn Algorithm that can handle infinite unlimited cooperative churn.*

Let us sketch the global solution. The algorithm chooses the coordinator, for example the process with the highest id, to handle churn requests. All processes know this coordinator and forward their requests to it. The coordinator serializes the requests handling. For each request, the coordinator sends the topology updates to the churning process and its neighbors. The coordinator waits for the process acknowledgements before starting the next request. If the coordinator x itself requests to leave, it stops handling other churn requests, selects the next coordinator y, forwards the incoming requests to y. The new coordinator y does not start handling requests until it gets the permission from x. Meanwhile, x informs all processes of the coordinator change, waits for their acknowledgements, forwards the permission for y to start handling requests and then leaves. This algorithm satisfies all the properties of the Fair Churn Algorithm.

Note that the outlined algorithm handles the least restrictive churn: infinite and unlimited. Therefore, this algorithm also handles infinite limited and finite limited and unlimited, see Figure 1. We now focus on local algorithms.

4 Linear Topology Churn Handling

Linear topology under churn. In a linear topology, each process p maintains two identifiers: *left*, where it stores the largest identifier less than p and *right*, where it stores the smallest identifier greater than p. Processes are thus joined in a chain. For ease of exposition, we consider the chain laid out horizontally with higher-id processes to the right and lower-id processes to the left. The largest process stores positive infinity in its *right* variable; the smallest process stores negative infinity in *left*. A *left end* of a link is the smaller-id neighbor process. A *right end* is the greater-id process.

As a process joins or leaves the overlay network, it may change the values of its own or its neighbors variables thus transitioning the link from one state

to another. In a linear topology, a link is *transitional* if its left end is not a neighbor of its right end or vice versa. The link is *stable* otherwise. The largest and smallest processes may not leave. The links to the right of the largest process and to the left of the smallest processes are always stable. A process may leave the overlay network only after it has joined. We assume that in the initial state of the overlay network, all links are stable.

```
constant p // process identifier
variables
    left, right: ids of left and right neighbors,
                 ⊥ if undefined
    leaving: boolean, initially false, read only,
                 application request
    busy: boolean, initially false; true when
          servicing a join/leave request
          or when joining
    C: incoming channel

actions
joinRequest:
    join ∈ C ⟶
        receive join (reqId)
        if (p < reqId < right) and not leaving
        and not busy then
            send sua(right) to reqId
            busy := true
        else
            if reqId < p then
                send join(reqId) to left
            else
                send join(reqId) to right

leaveRequest:
    leave ∈ C ⟶
        receive leave(reqId, q)
        if reqId = right and not leaving
        and not busy then
            send sua(⊥) to q
            busy := true
        else
            if p <= reqId then
                send leave(reqId, q) to left
            else
                send leave(reqId, q) to right

setUpA:
    sua ∈ C ⟶
        receive sua(reqId) from q
        if reqId ≠ ⊥ then // Join 1.1 received
            right := reqId
            left := q
            send sua(⊥) to right
        else // Join 1.2 or Leave 1 received
            left := q
            send sub to left

setUpB:
    sub ∈ C ⟶
        receive sub from q
        if q ≠ right then // Join 2.2 or Leave 2 received
            send tda to right
            right := q
        else // Join 2.1 received
            send sub to left

tearDownA:
    tda ∈ C ⟶
        receive tda from q
        if q ≠ left then // Join 3 or Leave 3.2 received
            send tdb to q
        else // Leave 3.1 received
            send tda to right

tearDownB:
    tdb ∈ C ⟶
        receive tdb from q
        if q ≠ right then // Join 4 or Leave 4.2 received
            send ftd to q
            busy := false
        else // Leave 4.1 received
            send tdb to left

tranDone:
    ftd ∈ C ⟶
        receive ftd from q
        if leaving then // Leave 5 received, p may exit
            right = ⊥
            left = ⊥
        else
            busy := false // Join 5 received
```

Fig. 2. Algorithm $\mathcal{CL}$ for process p.

Algorithm description. We present a local algorithm *Unfair Infinite Unlimited Churn* ($\mathcal{CL}$) that satisfies the four properties of the Unfair Churn Problem while handling unfair unlimited churn and maintaining a linear topology. The basic idea of the algorithm is to have the *handler* process with the smaller identifier of the place of join coordinate churn requests to its immediate right. This

handler considers one such request at a time. This serializes request processing and guarantees the accepted request's eventual completion.

The algorithm is shown in Figure 2. To maintain the topology, each process p has two variables: *left* and *right* with respective domains less than p and greater than p. Read-only variable *leaving* is set to **true** by the environment once the joined process wishes to leave the overlay network. Variable *busy* is used by the handler process to indicate whether it currently coordinates a churn request, or is initialized to **true** for a joining process. The incoming channel for process p is variable C. Processes do not accept churn requests when *busy* is **true**.

The request is sent in the form of a single *join* or *leave* message. We assume that a *join* and, for symmetry, a *leave* message is inserted into an incoming channel of an arbitrary joined process in the overlay network.

Message *join* carries the identifier of the process wishing to join the overlay network. Message *leave* carries the identifier of the leaving process as well as the identifier of the process immediately to its right. Actions *joinRequest* and *leaveRequest* describe the processing of the two types of requests. If the receiver realizes that it is to the immediate left of the place of join or leave, and the receiver is not currently handling another request, i.e. $busy \neq \textbf{true}$, and it does not want to leave, it starts handling the arrived request. Otherwise, the recipient process forwards the request to its left or right.

Request handling is illustrated in Figure 3. It is similar for join and leave and is divided into five stages. The first two stages are *setup* stages: they set up the channels for the links of the the joining process or for the processes that are the neighbors of the leaving process. The third and forth stages are *teardown* stages: they remove the channels of the links being replaced. The last stage informs either the leaving process that it may exit, or the joining process that it may start coordinating its own churn requests. In the case of join, links between two pairs of neighbors need to be set up, hence the setup stages are divided into two substages 1.1, 1.2, 2.1 and 2.2, and links between one pair of neighbors are tore down in stages 3 and 4. Similarly, in the case of leave, link setup stages 1 and 2 establish links between a pair of neighbors, followed by the teardown stages substages 3.1, 3.2, 4.1 and 4.2 to tear down links between two pairs of neighbors, then stage 5. We include the stage and substage numbers in the comments of Figure 2. The messages transmitted during corresponding stages are 1. set up A **sua**, 2. set up B **sub**, 3. tear down A **tda**, 4. tear down B **tdb** and 5. finish teardown **ftd**.

$\mathcal{CL}$ correctness proof. The formal proof is here [18] but the idea is as follows. We denote message **tda** or **tdb** as **td***. Similarly, **su*** is **sua** or **sub**. We show that in $\mathcal{CL}$, a teardown **td*** message is the last in the channel being torn down. Similar **su*** is the first message in a channel to be set up. The processes locally handle churn request sequentially. Thus, no regular messages are lost in the transition process. Moreover, the messages are eventually received and forwarded correctly, which leads to some churn request eventually being handled. Hence the below theorem.

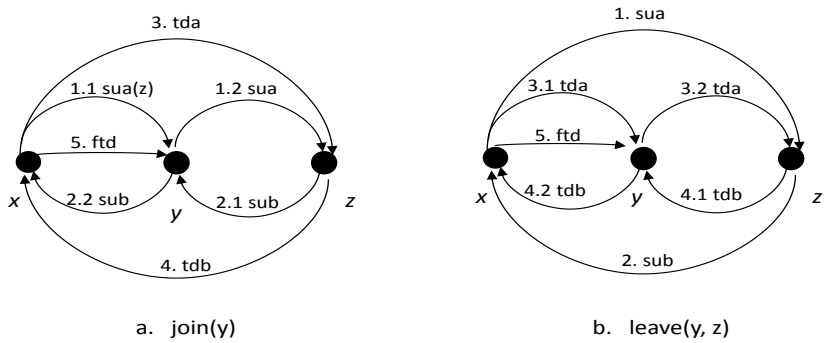

Fig. 3. $\mathcal{CL}$ join and leave request handling.

Theorem 3. *$\mathcal{CL}$ is a local Unfair Churn Algorithm that handles infinite unlimited churn and maintains the linear topology.*

Since finite churn limits the number of requests in a computation, it follows that $\mathcal{CL}$ handles finite unlimited churn and maintains the linear topology.

5 Skip List Churn Handling

In this section, we describe the algorithm $\mathcal{CSL}$ that handles unlimited infinite churn to maintain a deterministic 1-2 skip list. The advantage of a skip list over linear topology is that data search and churn request processing takes $O(\log N)$ steps compared to the linear search complexity. A *skip list* [32] consists of n levels with each level sorted in ascending order. The bottom level 0 contains all processes in the overlay network. In a 1-2 deterministic skip list, processes at level $i+1 > 0$ skip over one or two processes at level i. Algorithm $\mathcal{CSL}$ derives from $\mathcal{CL}$. Therefore, instead of presenting the code for the algorithm, we describe its operation. We use the same system model defined in Section 2.

Variables. Similar to the $\mathcal{CL}$ algorithm, variable *leaving* indicates if a process wishes to leave. At every level i, each process uses the *busy* variable to block itself at that level. Also, at every level i, each process x stores 1-hop and 2-hop neighbor identifiers. The first hop are *conversational* links that are used to exchange messages. The variables $x.i.l$ and $x.i.r$ store 1-hop left and right conversational neighbors. The second hop are *informational* links that do not contain messages but are used by the algorithm to make decisions. Variables $x.i.2l$ and $x.i.2r$ store 2-hop informational neighbors. Boolean variable $x.i.up$ indicates whether the process x exists at level $i+1$. If $x.i.up = \textbf{true}$, process x is *up* at level i, it is *down* otherwise. The smallest and largest id processes never leave and are present at every level of the skip list. The smallest process stores negative infinity in its $i.l$ and $i.2l$ variables. The largest process stores positive infinity in its $i.r$ and $i.2r$ variables.

Phases of operation. We use two phases to construct a 1-2 skip list: permission and construction. The *permission phase* gathers all necessary permissions and blocks all processes involved in a particular churn request from accepting additional churn requests. Once all permissions are gathered and the required processes are blocked by setting *busy* to **true** at each required level, the *construction phase* carries out the topology modification related to the churn request.

Permission phase. The permission phase proceeds recursively from level 0. At each level i, the handler considers the churn request. If it is not *busy* handling another churn request or wishing to leave, it blocks itself from considering any other requests, gathers the necessary permissions for the request at this level and, if necessary, submits the request to level $i+1$ and awaits level $i+1$'s permission. Once level i is secured, the permission is submitted to the lower level handler. If permission is not secured, a rejection is sent to the lower level handler.

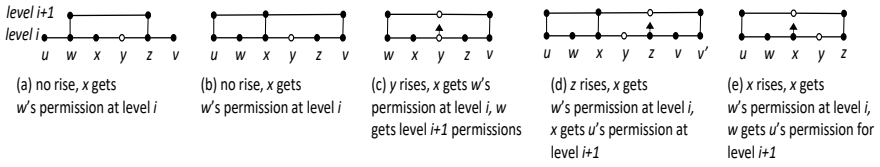

Fig. 4. The cases of process x coordinating y's joining.

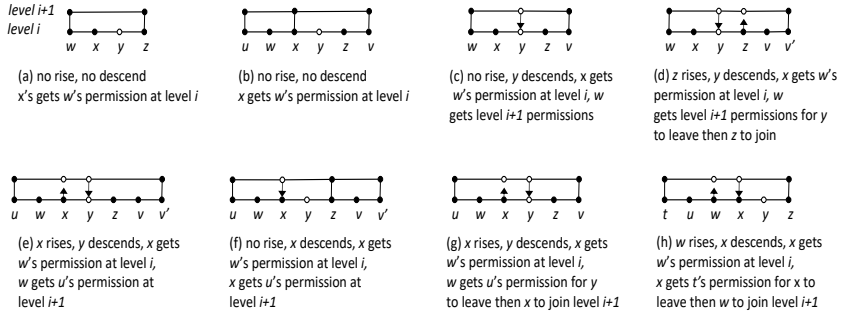

Fig. 5. The cases of process x coordinating y's leaving.

To determine the necessary permissions required for the join request from process y at level i, the handler process x considers the five cases shown in Figure 4. Similarly, for leaving, x considers the eight cases in Figure 5. For all cases, handler x at level i requests permission from its left neighbor w. Once

x gets the permission from w, if necessary, x requests permission from its left neighbor at level $i+1$. Before w replies to x, it may need to get permission from its left neighbor at level i, if further necessary, w requests permission from level $i+1$ becoming the handler at level $i+1$.

To determine the processes that must rise or descend, handler x requires 2-hop information, as opposed to only 1-hop information per the $\mathcal{CL}$ algorithm. Let's consider join Case e in Figure 4 in more detail. Process x is the handler and y is requesting to join. A hollow circle indicates a process whose status is changing at the corresponding level. At level i, w and x are *down* while u and z are *up*. When handler x accepts y's join request at level i, x examines its 2-hop neighborhood status and determines that it must rise and join level $i+1$ and that y's status must change to be *down*. Process x first requests w's permission at level i. If w is not blocked handling another request, w blocks itself. Then, w sends to u a request for x to join level $i+1$. Process u becomes the handler of the request at level $i+1$. If the necessary permissions are obtained at this and higher levels, w sends the permission to x and x sends it further downward. If the request is rejected, the process unblocks itself and sends the rejection downward.

Once the permission phase for a certain churning process y ends, the appropriate 2-hop neighbors are not able to join or leave. Indeed, if y is leaving, y, x and w are blocked. Process y rejects all requests from z, so z cannot leave. Moreover, since z forwards to y for (i) its right neighbor v to leave or (ii) a new neighbor z' to join, where $z < z' < v$, y's 2-hop right neighborhood is precluded from joining or leaving. The situation is similar if y is leaving.

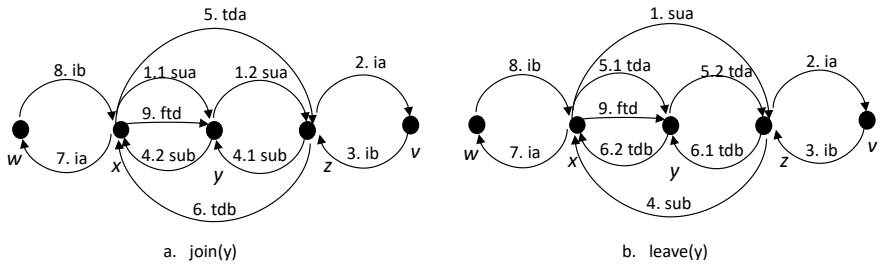

Fig. 6. $\mathcal{CSL}$ join and leave request handling.

Construction Phase. The construction phase proceeds from the top level down. At each level, the construction phase operates similar to $\mathcal{CL}$ algorithm. See Figure 6. The setup and tear down messages, **sua, sub, tda** and **tdb**, setup and tear down 1-hop conversational links. The informational messages, **ia** and **ib**, are added to maintain the informational 2-hop right and 2-hop left neighbor links and status, and to unblock y's 2-hop left neighborhood.

Correctness proof. The formal proof of $\mathcal{CSL}$ correctness is here [18]. The basic operation of the algorithm is similar to that of $\mathcal{CL}$, the major additions are the multi-level permission and construction phases. We show that for each request, the phases eventually end. Indeed, since the difference between the number of join and leave requests is bounded, the number of levels in a skip list is bounded also. The number of steps at each level is finite. Hence, eventually, the permission phase either returns the permission or rejection. The only way it can return a rejection is if some other request succeeds. Once all permissions are gathered, the construction phase proceeds in a similar manner. The correctness proof result is summarized in the below theorem.

Theorem 4. *$\mathcal{CSL}$ is a local Unfair Churn Algorithm that handles infinite unlimited churn and maintains a 1-2 skip list.*

Since finite churn limits the number of requests in a computation, it follows that $\mathcal{CSL}$ handles finite unlimited churn and maintains a 1-2 skip list.

6 Extensions and Future Work

Our solution for a 1-2 skip list can be extended to generalized skip lists and skip graphs. Notice, the locality of a 1-2 skip list is 2 and the permission phase blocked, whether explicitly or implicitly, the 2-hop neighborhood of the churning process. For a 2-3 skip list, the locality is 3, and the permission phase should block the 3-hop neighborhood. In general, the permission phase should block the specific neighborhood of the churning process. The construction phase should be modified to include the serialization of additional information messages, **ia*** and **ib***, to reach the specific neighborhood. For a 2-3 skip list, the informational messages are sent to the 2-hop and 3-hop neighbors of the churning process. The set up and tear down message patterns remain the same for the 1-hop neighborhood covering the conversational links.

As further research, it is interesting to consider extensions of $\mathcal{CL}$ to ring structures such as Chord [37] or Hyperring [5]. Another important area of inquiry is addition of limited adversarial churn.

References

1. David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP*, pages 131–145, New York, NY, USA, 2001. ACM.
2. J. Augustine, A.R. Molla, E. Morsy, G. Pandurangan, and P. Robinsonand E.Upfal. Storage and search in dynamic peer-to-peer networks. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 53–62. ACM, 2013.
3. J. Augustine, G. Pandurangan, and P. Robinson. Fast byzantine agreement in dynamic networks. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 74–83. ACM, 2013.

4. J. Augustine, G. Pandurangan, P. Robinson, S. Roche, and E. Upfal. Enabling robust and efficient distributed computation in dynamic peer-to-peer networks. In *Foundations of Computer Science, 2015 IEEE 56th Annual Symposium on*, pages 350–369. IEEE, 2015.
5. B. Awerbuch and C. Scheideler. The hyperring: a low-congestion deterministic data structure for distributed environments. In *SODA*, pages 318–327, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.
6. B. Awerbuch and C. Scheideler. Towards scalable and robust overlay networks. In *IPTPS*, 2007.
7. B. Awerbuch and C. Scheideler. Towards a scalable and robust DHT. *Theory of Computing Systems*, 45(2):234–260, 2009.
8. S.A. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.
9. A. Berns, S. Ghosh, and S.V. Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. In *Symposium on Self-Stabilizing Systems*, pages 62–76. Springer, 2011.
10. E. Caron, F. Desprez, F. Petit, and C. Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. *Parallel Processing Letters*, 20(1):15–30, 2010.
11. T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 322–330. ACM, 1996.
12. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
13. T. Clouser, M. Nesterenko, and C. Scheideler. Tiara: A self-stabilizing deterministic skip list and skip graph. *Theor. Comput. Sci.*, 428:18–35, 2012.
14. S. Dolev and R.I. Kat. Hypertree for self-stabilizing peer-to-peer systems. In *NCA*, pages 25–32, 2004.
15. M. Drees, R. Gmyr, and C. Scheideler. Churn-and dos-resistant overlay networks based on network reconfiguration. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), California USA*, pages 417–427. ACM, 2016.
16. D. Foreback, A. Koutsopoulos, M. Nesterenko, C. Scheideler, and T. Strothmann. On stabilizing departures in overlay networks. In *Symposium on Self-Stabilizing Systems*, pages 48–62. Springer, 2014.
17. D. Foreback, M. Nesterenko, and S. Tixeuil. Infinite unlimited churn (short paper). In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 148–153. Springer, 2016.
18. Dianne Foreback, Mikhail Nesterenko, and Sébastien Tixeuil. Churn possibilities and impossibilities. Technical Report hal-01753397, HAL, 2018.
19. D. Gall, R. Jacob, A.W. Richa, C. Scheideler, S. Schmid, and H. Täubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In *LATIN*, pages 294–305, 2010.
20. S. Gambs, R. Guerraoui, H. Harkous, F. Huc, and Anne-Marie A.-M. Kermarrec. Scalable and secure polling in dynamic distributed networks. In *31st Symposium on Reliable Distributed Systems (SRDS)*, pages 181–190. IEEE, 2012.
21. R. Guerraoui, F. Huc, and A.-M. Kermarrec. Highly dynamic distributed computing with byzantine failures. In *PODC*, pages 176–183. ACM, 2013.
22. T.P. Hayes, J. Saia, and A. Trehan. The forgiving graph: A distributed data structure for low stretch under adversarial attack. *Distributed Computing*, 25(4):261–278, 2012.

23. R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *PODC*, pages 131–140, 2009.
24. Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Skip+: A self-stabilizing skip graph. *Journal of the ACM (JACM)*, 61(6):36, 2014.
25. S.Y. Ko, I. Hoque, and I. Gupta. Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols. In *SRDS*, pages 259–268, 2008.
26. A. Koutsopoulos, C. Scheideler, and T. Strothmann. Towards a universal approach for the finite departure problem in overlay networks. In A. Pelc and A.A. Schwarzmann, editors, *SSS*, pages 201–216. Springer, 2015.
27. Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. Towards worst-case churn resistant peer-to-peer systems. *Distributed Computing*, 22(4):249–267, 2010.
28. Giuliano Mega, Alberto Montresor, and Gian Pietro Picco. On churn and communication delays in social overlays. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 214–224. IEEE, 2012.
29. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.
30. R. Mohd Nor, M. Nesterenko, and C. Scheideler. Corona: A stabilizing deterministic message-passing skip list. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 356–370, Grenoble, France, October 2011.
31. R. Mohd Nor, M. Nesterenko, and S. Tixeuil. Linearizing peer-to-peer systems with oracles. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS, Osaka, Japan, November 13-16, Proceedings*, pages 221–236, 2013.
32. William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
33. Christina Rickmann, Christoph Wagner, Uwe Nestmann, and Stefan Schmid. Topological Self-Stabilization with Name-Passing Process Calculi. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
34. A.I.T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350. Springer, 2001.
35. Jared Saia and Amitabh Trehan. Picking up the pieces: Self-healing in reconfigurable networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.
36. A. Shaker and D.S. Reeves. Self-stabilizing structured ring topology p2p systems. In *Fifth International Conference on Peer-to-Peer Computing (P2P)*, pages 39–46. IEEE, 2005.
37. I. Stoica, R.Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.
38. D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *IMC*, pages 189–202, October 2006.