Infinite Unlimited Churn (Short Paper)

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

[1] Kent State University, Kent, OH, USA
dforebac@kent.edu
[2] UPMC Sorbonne Universités and IUF, Paris, France

Abstract. We study unlimited infinite churn in peer-to-peer overlay networks. Under this churn, arbitrary many peers may concurrently request to join or leave the overlay network; moreover these requests may never stop coming. We prove that unlimited adversarial churn, where processes may just exit the overlay network, is unsolvable. We focus on cooperative churn where exiting processes participate in the churn handling algorithm. We define the problem of unlimited infinite churn in this setting. We distinguish the fair version of the problem, where each request is eventually satisfied, from the unfair version that just guarantees progress. We focus on local solutions to the problem, and prove that a local solution to the Fair Infinite Unlimited Churn is impossible. We then present our algorithm $\mathcal{UIUC}$ that solves the Unfair Infinite Unlimited Churn Problem for a linearized peer-to-peer overlay network. We extend this solution to skip lists and skip graphs.

1 Our Contribution

We study the problem of churn in (structured) peer-to-peer overlay networks in the asynchronous message passing system model. Peers in the overlay network maintain the identifiers of its overlay neighbors in memory; message routing is left to the underlay. Specifically, we consider cooperative churn as opposed to adversarial when processes just exit. We prove that there does not exist an algorithm that can handle unlimited adversarial churn.

We define infinite unlimited churn in peer-to-peer overlay networks. *Infinite churn* handles an unbounded number of churn requests under which the overlay network has to maintain services (e.g. content retrieval) while handling it. This is opposed to *finite* churn, where services are either considered suspended or they are disregarded altogether [2]. We consider *unlimited churn* where there is no bound on the number of concurrently joining or leaving processes; potentially all processes presently in the overlay network may request to leave concurrently. Note that the infinite and unlimited churn properties 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. To the best of our knowledge, this paper is the first systematic study of unlimited infinite churn.

B. Bonakdarpour and F. Petit (Eds.): SSS 2016, LNCS 10083, pp. 148–153, 2016.
DOI: 10.1007/978-3-319-49259-9_12

2 The Infinite Unlimited Churn Problem

We distinguish fair and unfair types of the problem. A request to join and, in cooperative churn, leave the overlay network is submitted to the overlay by the *churning process*. A churn handling algorithm is *fair* if it eventually satisfies every request. By contrast, a churn algorithm that allows the possibility, under infinite churn, to bypass indefinitely some requests (still guaranteeing progress, i.e., satisfying some churn requests indefinitely), is *unfair*. Unfair algorithms are possibly more efficient.

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 the infinite churn problem conforms to a combination of 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.

We define two variants of the problem. *The Unfair Infinite Unlimited Churn Problem* is the combination of request progress, terminating transition, message progress and message safety properties. *The Fair Infinite Unlimited Churn Problem* is the combination of fair request, terminating transition, message progress and message safety properties. In other words, Fair Infinite Unlimited Churn guarantees that every churn request is eventually satisfied while Unfair Infinite Unlimited Churn does not.

3 Impossibilities

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 to increase the distance between x and y by at least one. This constant m is the *expansion vicinity* of the topology. 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 request may potentially be far away, i.e. a large number of hops, from the place where the topology maintenance operation needs to occur. We will

consider an overlay network that maintains a linear topology, i.e., a topological sort. *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 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 minimum 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 be greater than the locality of the algorithm.

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

Theorem 2. *There does not exist a local solution to the Fair Infinite Unlimited Churn Problem for an expansive overlay network topology.*

```
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. 1. Algorithm $\mathcal{UIUC}$ for process p.

[1] Proofs and referenes are in the full version of the paper [3].

4 Local Unfair Infinite Unlimited Churn ($\mathcal{UIUC}$)

We present a local algorithm *Unfair Infinite Unlimited Churn* ($\mathcal{UIUC}$) in Fig. 1 that satisfies the four properties of the Unfair Infinite Unlimited Churn Problem while maintaining a linear topology. The basic idea of the $\mathcal{UIUC}$ algorithm is to have the *handler* process with the smaller identifier 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 request is sent in the form of a single *join* or *leave* message. Each process p maintains two identifiers: *left*, where it stores the largest identifier greater than p and *right*, where it stores the smallest identifier less 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 it is initialized to **true** for a joining process. The incoming channel for process p is variable C. Communication channels are FIFO with unlimited message capacity. We refer to processes and their identifiers interchangeably. Process p is a *neighbor* of process q if q stores the identifier of p. Note that q is not necessarily a neighbor of p. A process may send a message to any of its neighbors. A process may send a message only to the receiver with a specific id, i.e., we do not consider broadcasts or multicasts. The processes have unique identifiers. 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.

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. If the receiver realizes that it is to the immediate right of the place of join or leave, and the receiver is not currently handling another request, i.e., $busy \neq$ **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 accomplished by the order messages are sent and received to setup stable links and tear down transitional links to maintain the linear topology. 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 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, two links need to be set up, hence the setup stages are divided into two substages 1.1, 1.2, 2.1 and 2.2, followed by the teardown stages 3 and 4, then stage 5; these join substage numbers are included in the comments of Fig. 1. Similarly, in the case of leave, links setup stages 1 and 2 are followed by the teardown stages that are divided into substages 3.1, 3.2, 4.1, 4.2 because two links need to be torn down, then stage 5. 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**.

Theorem 3. *Algorithm $\mathcal{UIUC}$ is local and it solves the Unfair Infinite Unlimited Churn Problem.*

5 $\mathcal{UIUC}$ Extensions to Skip List and Skip Graph and Further Work

Churn algorithm $\mathcal{UIUC}$ extends to more complicated topologies such as skip lists and skip graphs In these topologies, the processes have links on multiple levels. The processes are linearized in the lowest level. In the higher levels, the processes have links to progressively more distant peers. These higher level links accelerate overlay network searches and other operations. To extend $\mathcal{UIUC}$ to such a structure a separate version of $\mathcal{UIUC}$ should be run at each level. The churn request should bear the level number to differentiate which level $\mathcal{UIUC}$ they belong to. The churning process should proceed up and down the levels as follows. A joining process first joins the first, linear, level, then the next and so on until it joins all the levels appropriate to the particular structure. The leaving process should proceed in reverse: the leaving process requests to leave the levels in decreasing order.

As further research it is interesting to consider extensions of $\mathcal{UIUC}$ to ring structures such as Chord [4] or Hyperring [1]. Another important area of inquiry is addition of limited adversarial churn. This problem is difficult to address in the asynchronous message passing model where the exited process may not be differentiated from a slow one. Oracles determining a process exit [2] may have to be used.

References

1. Awerbuch, B., Scheideler, C.: The hyperring: a low-congestion deterministic data structure for distributed environments. In: SODA, pp. 318–327. Society for Industrial and Applied Mathematics, Philadelphia (2004)
2. Foreback, D., Koutsopoulos, A., Nesterenko, M., Scheideler, C., Strothmann, T.: On stabilizing departures in overlay networks. In: Felber, P., Garg, V. (eds.) SSS 2014. LNCS, vol. 8756, pp. 48–62. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11764-5_4

3. Foreback, D., Nesterenko, M., Tixeuil, S.: Infinite unlimited churn. Technical report 1608.00726, arXiv (2016)
4. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Frans Kaashoek, M., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for Internet applications. IEEE/ACM Trans. Netw. **11**(1), 17–32 (2003)