By how far have recent churn-reduction techniques improved quality of experience in peer-to-peer video streaming environments?

Martin Symons

May 2, 2024

Abstract

abstract !!!

1 Introduction

Within the past ten years, video streaming traffic across the internet has exploded. Traditional client-server architectures places a large load on the small portion of nodes controlled by the streaming host, and lead to dramatic infrastructure costs. To combat this, peer-to-peer systems have been proposed that spread usage across the network's collective resources. Starting with Coolstreaming in 2004, these networks became defacto for IPTV streaming, especially pirated streams. With their utility proven, research accelerated, in pace and complexity. Industry, however, did not pick up. Since the shutdown of New Coolstreaming and PPLive in the late 2000's, these newer solutions have seen little real-world usage.

As such, the actual impact of current research on network performance is vague. Still, research since has suggested that churn has a substantial and exponential impact on client quality-of-service (QoS) beyond that which was considered in these original models. This paper investigates several marked improvements over the best-documented benchmark architecture, New Coolstreaming, with particular focus on methods to reduce churn throughout the network. wordsw words words

2 Literature Review

2.1 Peer-to-Peer Streaming Fundamentals

In the late 1990's the traditional client-server architecture began to crack, as computer capabilities began to outpace bandwidth growth across the internet. Small clusters of server nodes were expected to take the burden, whilst ample client resources waited idle. BitTorrent allows a swarm of peers to collaborate in file transmission, maximizing global utilization without overwhelming any single node. Peers instead connect to a tracker or query a global DHT, usually *Mainline DHT*, to gather a list of participatory nodes. Targetting the rarest blocks first, data chunks within the swarm are piped directly over UDP node-to-node. As more peers join the network, the overall bandwidth increases, with download speeds following suit.

This approach exploded onto the internet, claiming 35% of all upstream traffic (**archiveCacheLogicResearch**). It remained in the number one spot for total traffic until recently - finally dethroned in 2024 by video streaming.

In a file-share network QoS is taken mostly with upload speed, and by extension bandwidth utilization. Any intermediate action from loading a *.torrent* to completion is irrelevant. Video streaming, in contrast, introduces a large number of real-time requirements essential for a good user experience.

Peer-to-peer networks pipe data directly between nodes through a network, contrasted with the traditional client-server architecture seen most often today. Compared to file-sharing applications, peer-to-peer video networks face several new complexities. In the former, QoS is ruled almost entirely by bandwidth utilization. Users care only that a desired file eventually reaches them; the journey it takes to completion is unimportant. probably list some examples Video livestreaming, however, introduces new complexities. playoutdeadlines startupdelay end-to-enddelay one proposed solution was ip multicasting. it was . fucked. s owe don't use it and instead use the various better solutions

2.2 Performance Heuristics and Churn

explaining why we use/care about churn vs other characteristics

2.3 Overlay Structure

2.4 Peer Selection Strategies

2.5 Chunk Schedulers

2.6 Historic Implementations

We now consider historic implementations of peer-to-peer streaming systems in relation to the above characteristics. Of most relevance to our paper is DONet, commercially branded Coolstreaming (**1498486**). Considered the first of its breed, Coolstreaming Xie et al. 2007 **asdasd 1498486**

2.7 Implementation-Specific Improvements

3 Methodology

We aimed to pit three models against each other in various QoS tests to gather specific numbers on the improvements between each. Initial tests would be run on New Coolstreaming, widely considered the last popular IPTV P2P system and a good benchmark . We would then extend this with fitting modular improvements for further measurements. were particular targets, because . Finally, we expected to implement a monolithic model, , to compare the capacity of incremental improvements versus the design of a single, deeply-coupled architecture.

As a simulation environment, we chose OverSim. Its included churn mechanisms, quick-switching between simplified and realistic underlay models, and complete debugging suite eased the otherwise involved development cycle. Ejecting from OverSim to OMNET++ would also have been trivial, though was never necessary.

Statistics would be presented with the built-in OMNET++ visualization tools.

4 Implementation

We based our initial experiments on New Coolstreaming as described in B. Li et al. 2008. We quickly ran into problems - whilst the paper describes the stream manager and buffer map exchange in great detail, little time is given to the membership and partnership managers. The upkeep of the *mCache* with incoming peers is unspecified, as is most connection management action

which?

we probably already explained this in the lit review

name some shit

blah

name

needs expansion. we could mention dates?? what

related to churning or failing nodes and some key equations to system function. We pushed forward and attempted to fill the blanks ourselves; the final result, whilst technically functional, invariably failed to meet playout across nodes and was in no way correspondent of New Coolstreaming's measured real-world performance.

The New Coolstreaming paper concludes its discussion on the problem modules stating *"these basic modules form the base for the initial Coolstreaming system,"* and that the New Coolstreaming *"has made significant changes in the design of other components."* We thus considered that these modules were holdovers from the older design, implying New Coolstreaming must be built with DONet/Coolstreaming as groundwork. We thereby set about an implementation of this more primitive design.

The final DONet implementation completed following two weeks of work. This network was similarly not ideal, though the cause was mostly banal - New Coolstreaming strips the buffer, scheduler and related messaging completely, so we saw no need to optimize these components. More worryingly, the partnership manager collapsed quickly under even minimal churn, discussed later in . Still, this constituted enough the groundwork needed to continue.

Returning with wiser eyes, we found that our architecture did not align at all with the basic modules in New Coolstreaming. *How could this be?* As discussed later in section 6.6, DONet has clarity problems of its own when describing parts of other systems, and we noticed that the output of these components - M-number exchanging partners ready for video transmission - *did* align with the older model. We therefore treated this as a simple faulty description, and moved on to the design of the stream manager.

The well-specified stream manager came through without a hitch, but placed new constraints on the partnership manager that our already brittle implementation could not bear. We hence designed *Partnerlink*, a relationship algorithm reconciling the high-churn overlay with New Coolstreaming's low-churn subscription requirements and performance at scale. This new algorithm meshed well with the stream manager, and brought our implementation to a close.

The full development process took over a month. We were therefore not able to complete any further models or make any comparisons on QoS benefits.

5 Results and Analysis

In testing, we begin with two origin nodes and join 30 client nodes over the course of 90 seconds. Nodes live on average 500 seconds before churning; we allow 300 seconds for the overlay to stabilize before a 2000 second measurement period. The 500Kbps source video is split into 10 substreams, containing blocks of one second duration. Partner and membership count M is limited to 15.

We first test our model on a simple underlay without packet dropping, bandwidth limitations or timeouts. We find that our partnership mechanism works well in this environment, averaging 14.89 partners across all nodes over the entire measurement period. No node is ever seen with below 11 partners. We also, however, find our potential partner count M_c to be the same value, as our failure mechanisms are almost never engaged in this ideal environment. We find that our model maintains a satisfactory number of parent-child relationships with an average of 9.3. Despite this, our playout is poor: only 61.58% of blocks are received before playout time. Our efficiency is reasonable - of all received blocks, 1.92% are duplicates and 4.645% are outside the bounds of the receiver's buffer, almost all being too old.

there is a fucking term for this i know

We test again on a realistic underlay simulating the best-effort internet, with bandwidth limitations, packet loss and queuing. Under this environment, the partnership mechanism begins to fail: despite recording an average M_c of x, our actual partner count averages y. We explore the origin of this and the resultant changes made to *Partnerlink* in section 7.5. Surprisingly, our playout does not completely fail in turn -

6 Discussion

What went wrong? New Coolstreaming is not unique as an overlay; no features within should prove particularly challenging to implement. In this section, we explore the Coolstreaming family as a whole, and illuminate the many challenges faced in their implementation amiss in the papers themselves.

6.1 The Coolstreaming Family Tree

We have so far regarded Coolstreaming as a dyad of the mesh-pull DONet/Coolstreaming and hybrid-push-pull New Coolstreaming, proposed across two papers. The reality is not so simple. Coolstreaming is formed of two models, as described.

However, they have been proposed under *four* different names across *four* papers:

- As *DONet* and *Coolstreaming*, in *CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming* Zhang et al. 2005
- As *Coolstreaming*, in *Coolstreaming: Design, Theory, and Practice* Xie et al. 2007
- As *Coolstreaming+*, in *An Empirical Study of the Coolstreaming+ System* Bo Li et al. 2007
- As *"the new Coolstreaming,"* in *Inside the New Coolstreaming: Principles, Measurements and Performance Implications* B. Li et al. 2008.

Note that the name *Coolstreaming* is intended by this final paper, but *The New Coolstreaming* became the colloquial model name as a result of its title. Only the first paper contains the old model we have discussed as *DONet/Coolstreaming* - the others all specify *New Coolstreaming*, despite the name differences.

The name *Coolstreaming* legally refers both to the older *DONet* model and the newer *New Coolstreaming* model. The confusion that results is obvious. Kondo et al. 2014 describes the *SCAMP* membership protocol, the push-pull mechanism and the bootstrap node as part of the same model, despite *SCAMP* being specific to the mesh-pull *DONet*. Beraldi, Galiffa, and Alnuweiri 2010 makes much the same error. Lan et al. 2011 takes *DONet* as its key example of a buffer-map driven overlay, but ascribes it the synchronization method seen in *New Coolstreaming*. Further examples still can be found of correct prose, but with Xie et al. 2007 being cited in Zhang et al. 2005's place, or vice versa.

It is interesting to note that the papers themselves appear to have issues keeping the versions straight: B. Li et al. 2008 describes the stream manager and new mCache system as part of the *"initial Coolstreaming system"* and replaced in *"the new Coolstreaming system"*, despite all of these components being local to *New Coolstreaming* only - hence our initial hesitance to visit *DONet*.

This escalates considering the final three papers. Xie et al. 2007 is the canon definition of *New Coolstreaming*, alongside analysis of a real-world test period to determine convergence rates, start-up delay and other overlay-specific statistics. The other papers duplicate this and add further analysis: Bo Li et al. 2007 splits users into categories, identifying network traversal problems and their respective impact on contribution. B. Li et al. 2008

performs an additional simulation to identify ideal values for key system parameters.

This duplication means each paper opens with a definition of *New Coolstreaming*. These are not summaries - the *Coolstreaming+* specification is shortest yet still clocks in at two and a half pages. The majority of this text is word-for-word identical between papers, or at best reworded. The membership and partnership managers, though, have their specifications cut down completely, no longer parsing as a working system. For instance, connection management is resolved in Xie et al. 2007 by an off-handed mention of TCP - which includes a leaving and timeout mechanism, if specified. In the other papers, TCP is never mentioned; no other connection management is described. Mechanisms to fill the bootstrap node's *mCache* alongside one key function related to playout initialization are similarly constrained to this paper, despite these papers claiming to *"describe the architecture and components in the new Coolstreaming system"*.

Luckily, we appear to be the only researchers to fall into this trap. Resulting development time was certainly extended, but our criticisms of the paper and our final solution remain valid.

6.2 What does Coolstreaming need to be?

flow this

The *Coolstreaming* family has been replaced in production by monolithic solutions like . *Coolstreaming*'s usage now lies entirely in the research domain - used as a base for reproducible results, a testbed for new modular features or a point of comparison between larger monolithic models. In all of these cases, consistency and ease of development take priority over QoS performance.

mention wio-ta-diof

The issues mentioned so far are a sign that *Coolstreaming* was never designed for this purpose. *Coolstreaming* aimed to be the leading production overlay of its time, including optimizations that were inconvenient but deemed worth the miniscule performance boost. More than this, *Coolstreaming* was developed as a commercial system first and a research paper second. This may explain the missing features or heavy modifications of existing solutions - the system was still under active maintenance as the papers were written, and so work had to be quick moreso than correct.

In the next section, we introduce a new model to the *Coolstreaming* family designed for research purposes, outlining our priorities and rationale in doing so.

6.3 The Fully-Connected Network Problem

Both *DONet* and *New Coolstreaming* make the same demands on the partnership manager: given a random partial view of the network, create and maintain at most M connections ready for video transmission. Neither paper provides any exact information on how this should be done.

Suppose a node n joins a network of size $M + 1$. All existing nodes in the network are fully saturated with partner count M. How can this node join the network?

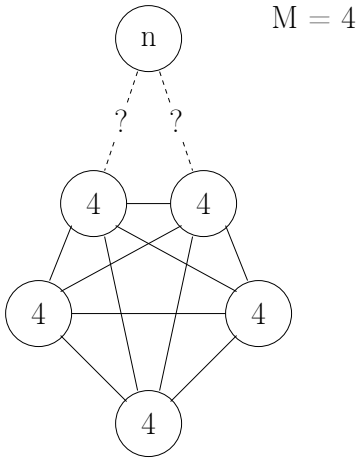

Figure 1: Node n cannot easily join this fully-connected network.

In our early *DONet* implementation, we forced nodes to accept all incoming partnerships, removing an old partnership to make room. This created long chains of dropped-and-replaced partnerships before the overlay settled and worsened the impact radius of churn. The move to *New Coolstreaming* came with the demand to maximize partnership lifetimes, as churn would interfere with the push mechanism and result in wasted blocks being sent to a peer, totally invalidating this approach.

Several solutions exist, though none are trivial. Our final implementation requests a node to break one partnership, placing the new partner in the middle. This has a great number of implications on the network, most notably that M must now be divisible by 2. Since *DONet* uses $M = 15$ as example, this cannot be the implementation as intended by the original authors.

We can at least infer some details from *DONet*. The *mCache* data structure includes *num_partners* for each visible node, which is updated but never apparently used. In a healthy network, the likelihood of a node directly knowing another node with less than M partners is low. Instead, we assume this to be part of a gossiping mechanism where a gossiped partnership request is

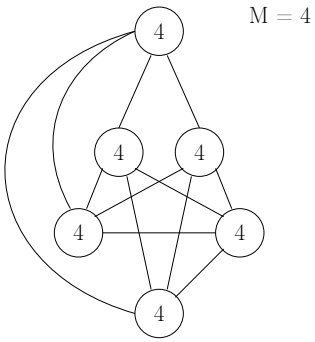

Figure 2: A solved topology, as according to the pairing method.

directed towards nodes with missing partners, which would also justify the use of *SCAMP* as opposed to a more standard partial view aggregator. We include a similar strategy as part of our implementation; we discuss this in detail later in section 7.3.3.

That said, the phrasing of the scheduled switching as established *"with nodes randomly selected from [the] mCache"* implies a more direct communication. In this case, it is unknown how *DONet* or *New Coolstreaming* solve this problem.

6.4 Choosing New Partners

The two inequalities measuring connection quality are also applied to any candidate partners following a detected failure. Inequality 1 limits the gap between a node's buffers and the parent's. If we apply this to candidates after failing because of this inequality, we will filter out any nodes further ahead than our old node, slipping further back in our buffer. Assuming we never change our playout position, this new node will not be able to service our entire buffer area.

Furthermore, the nodes seen within this inequality will mostly be those similarly falling behind the swarm; that is, nodes with equally poor connections. This groups poorly performant nodes and drags them backwards across the buffers, eventually collapsing entirely. In our model we disabled inequality 1 when measuring candidates, and performance resumed - *New Coolstreaming* must use some other unspecified mechanism to avoid this.

6.5 Requesting the Block

New Coolstreaming specifies an algorithm to compare all buffer maps from a node's first partners and calculate the to request. The method through which

this is done is somewhat obscure - the subscription map only communicates on-off booleans. We assume this is achieved by manipulating the latest blocks shown to each node to suggest the node has already received up to that block, despite having just joined. This requires some careful planning to execute without triggering subscriptions from nodes but is a clever usage of the limited data structure at play.

A feature unmentioned here is the behaviour-of and restrictions-on the playout index. Neither *Coolstreaming* implementation specifies a recovery mechanism - nodes are assumed to never fall completely behind the swarm. This leaves us two options - either the playout index must never buffer or pause after sync, and continue to play forward on encounter with missing blocks, or the recovery mechanism is left undefined. Given that there is a startup delay listed amongst the results, we can assume the latter. If the stream does include buffering or the node is suffering poor local performance, though, the need to recover is inevitable. There are many options for approach that impact performance, and one best solution should ideally be provided.

6.6 Production Optimizations

DONet defers to *SCAMP* for membership connection management, before overruling it on several counts. Where *SCAMP* automatically adjusts membership count in sync with the size of the system, *DONet* enforces a hard limit of M nodes, usually 15. The indirection mechanism is implemented only in primitive form, with only nodes within the origin's view being available as contacts. As *DONet* only uses *SCAMP* to gather a random partial view, and does not use it as a means of message transmission itself, *DONet* does not require *SCAMP*'s full set of guarantees. Whilst we have not been able to perform experiments to prove this, we hypothesize that these restrictions therefore make no negative impact on the network, included to reduce message overhead.

we need to establish ahead of time the fanout proofs of SCAMP etc.

Other optimizations are less successful. *SCAMP* specifies a leasing system to remove dead nodes from the pool, which *DONet* accepts. However, in the case that the partnership manager finds a partner that fails to send a buffer map in a given period, the partnership overrides the membership manager to remove the node from the *mCache*. The partnership manager then emits a departure message on its behalf, which is gossiped *"similarly to the membership message,"* i.e. once to a peer who then repeats it to every node in its *mCache*. Since exactly M departure messages are gossiped, every message would have to reach a peer knowing the failed node for a clean exit. The layers of indirections involved in membership management, however,

make it unlikely for this message to reach any such peer at all.

This mechanism is applied even to departure messages sent by a leaving node itself. *SCAMP* notes that each node knows the peers it is subscribed to through its *InView* list - thus, the only requirement for a clean exit is to send a departure to each node it contains. The gossiped alternative makes no gain in message overhead whilst almost totally invalidating its effect on network topology.

This appears to be a malformed optimization to allow nodes to depart each other in case of failure. Even if this were successful, the already-established leasing system means any performance gain would be negligible.

Successful or not, these changes represent a problem: we fall back to some existing research solution for a problem and then brain slug it for some slight gain, in the process tampering with that solution's proven performance. This is convenient for the original researcher, but requires careful cross-referencing of the two papers for accurate reproduction. New implementation of the model therefore becomes strenuous, and inaccuracies made much more likely.

is this actually a good turn of phrase

we need to flow this

7 Introducing *coolstreaming-spiked*

coolstreaming-spiked is a new iteration of *Coolstreaming* bringing the two models into synthesis. Research usage is targeted through an easier implementation, thorough central specification and compatibility with a wide range of IPTV improvements. Specifically:

- The SCAMP protocol is retained for backwards compatibility with research performed on the original *DONet*.
- The partnership connection algorithm is solved and documented as *Partnerlink*, a novel approach providing realistic performance and absolutely minimal churn impact for networks of any scale.
- Production optimizations are stripped back to focus on a smaller scope.
- Other fixes, such as the disabling of inequality 1 on candidates, are implemented as standard.

We now proceed to describe the function of this model.

7.1 Architecture

Our component architecture is identical to that seen in *New Coolstreaming*: a membership manager provides the model with a continuous pool of random nodes. The partnership manager filters these nodes to find connections

suited for video transmission, and handles the exchange of buffer maps. The stream manager creates parent-child relationships, owns the buffer and forwards blocks to peers. Each node within the architecture is provided a unique *NodeID* - an IP address will suffice.

We incorporate the substream system as defined in *New Coolstreaming*. The video sequence is split into blocks of equal size each assigned a timestamp sequence number. These are shared equally between K-number zero-indexed substreams, where the i-th substream contains blocks with sequence numbers $(nK + i)$ where n is a positive integer from zero to infinity. For instance, assuming $K = 4$, substream 2 would contain blocks with ID $\{2, 6, 10, 14, ...\}$.

7.2 Membership Manager

The membership manager makes the initial contact to an origin node. We then subscribe as members to a subset of nodes using SCAMP to provide the partial view. No changes are made to SCAMP - the standard subscription, unsubscription, indirection, leasing and recovery methods all apply. There is no limit on the size of the *mCache*. The partnership manager can no longer directly influence the mCache or send unsubscription messages on behalf of another node. These changes allow researchers to follow a well-specified, battle-tested paper for gossiping support without cross-referencing our own. The main output of this component is an arbitrarily sized set of random nodes across the network.

SCAMP does not describe any starting topology. A bootstrapped SCAMP network must contain at least two nodes engaged in a shared subscription.

In the case that the node ever becomes completely isolated and can no longer aggregate members, the node should recontact the origin and gossip a fresh subscription message.

7.3 Partnership Manager and *Partnerlink*

The partnership manager takes these nodes and begins to form TCP connections ready for block transmission. TCP performs the majority of connection upkeep on our behalf - if the connection ends, times out or fails, the partnership should be considered *failed*.

this doesn't mean anything

The buffer map system remains as specified in *New Coolstreaming*. A buffer map comprises two tuples of length K - the first listing the highest block sequence number received in each substream, the second listing the subscription of substreams to the receiving partner. For instance, a node receiving tuples of $\{40, 41, 42, 39\}$ and $\{0, 0, 0, 1\}$ from a partner should infer the node to have most recently received block 39 on substream 3, and prepare

a subscription for the node on that substream according to the subscription map. We also transfer the node's current *panic status* and a *NodeID* representing an *associated peer*, discussed later in 7.3.3 and 7.4. A node receiving a buffer map should store these values alongside their relevant partner; the latest block should only be set if the subscription is new as of this message, as we update this value locally as part of the stream manager. If a partner does not transfer a buffer map in some specified length of time, the partner is missing and the partnership should be considered *failed* - the partner connection is removed and M_c (discussed later) is reduced by 1.

7.3.1 *Partnerlink* Description

We now describe the *Partnerlink* algorithm. We first make two constraints on the system - a maximum number of partners at each node M is introduced, where $M \mod 2 = 0$. Our base join operation grants a node two links per request, so an even partner cap is essential. We define a starting network as two or more nodes in a valid system topology, since at least two peers are necessary to begin splitting. This basic network could be comprised of two guaranteed-trustworthy origin nodes, or built cooperatively with potentially-malicious client nodes.

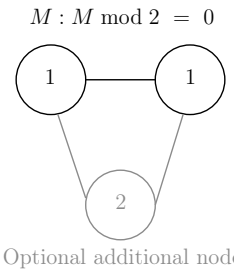

Figure 3: The base topology from which *Partnerlink* must operate.

Each node tracks an additional set of variables. We maintain a set of *locks* against *NodeID*s for which operations are still in progress, to protect the system when a node receives several requests regarding the same node. We also maintain an *anticipated* partner count M_c, tracking what our partner count will become if all ongoing operations resolve successfully. Making decisions against this value rather than our actual partner count ensures we do not accidentally request partners above our hard limit M.

Partnerlink operations can be split into three categories:

- **Initiation**, allowing a healthy node to create new connections with known nodes.

- **Management**, allowing the overlay to adapt after a failed partnership.
- **Panic**, allowing an unhealthy node to recover back to M connections.

We now discuss the systems and messages in place to allow for these operations.

7.3.2 *Partnerlink* Initiation

A node n entering the network for the first time awaits the first *SCAMP Inview* message to initialize the *Partnerlink* manager. A list of up to $M/2$ candidate partners is requested from the contained node, retrieved at random from its *mCache*. This is necessary as the local node's *mCache* needs time to fill with peers; relying on it for first partners comes with substantial startup delay. Upon reception at node n, these nodes are all locked and sent a *Split* message. This performs the split operation discussed prior, attempting to place the node in-between an existing partnership. The node also sets M_c to $M_r * 2$, where M_r is the number of nodes received from the initial exchange. A node a receiving a *Split* message checks that no partnership nor lock is held against n - if not, the node forwards a *TrySplit* message to a random unlocked partner b, holding locks against it and n. Node b performs the same checks, additionally requiring that a is not locked. If the checks pass, it returns a *TrySplitSuccess* message, breaks the connection with its old partner and initiates with the new node. Node a receiving a *TrySplitSuccess* breaks the connection similarly and unlocks the two nodes, after which the exchange is complete. Upon connection, node n removes its lock on node a. The newly joined node has now gained two links, breaking only one in the process. If all *Split* messages respond successfully, the node will have created exactly M partnerships.

If node b fails its checks, it responds a *TrySplitFailure*. If node a receives a *TrySplitFailure*, fails its own checks or does not know another unlocked partner, a *SplitFailure* is forwarded back to the new node n which reduces M_c by 2 and unlocks the contacted node. This causes the node to immediately begin panicking.

7.3.3 *Partnerlink* Panicking

If a node does not satisfy $M_c = M$, it is described as *panicking*. A node checks its panic state after each update to M_c. We begin to look for alternative ways to create new links in the system.

If $M_c = M - 1$, we cannot fill our empty partners by splitting, as we would breach our limit. We instead gossip a *Panic* message containing our

NodeID to a random partner. This message is directed across the network towards panicking nodes with priority, as determined from their buffer maps. Otherwise, the message is directed towards any node that is not the original panicking node or the last hop. Otherwise, the message is directed anywhere, or else dropped. If a panicking node receives a *Panic* message, it initiates a connection with the contained node and increases its M_c by 1. The original node receiving this connection does the same, unless it has since stopped panicking, in which case it rejects the connection.

This joins two panicking nodes together directly and grants one new link each. This is mostly effective in very small networks, where nodes impacted by an improper exit are very few hops together. As networks grow and stabilize, the hops between panicking nodes will increase, worsening delay before node recovery. Note that locks do not influence the transmission of this message, as its result on a partner does not influence a node's local link count.

If $M_c = M - 2$, we fall back to the splitting method. We gossip a *PanicSplit* message containing our *NodeID* to a random partner. This message contains a field *LastHopOpinion*, initialized to *CANT_HELP*. The same direction mechanism is applied, although with priority now given to non-panicking nodes and a ban on gossiping to locked partners. If a node receives a *PanicSplit* with *LastHopOpinion CANT_HELP*, the node performs similar checks as in a regular *Split* - that we are not already partnered with nor hold any locks against the contained node. If the checks pass, the message is gossiped to a random unlocked partner with a *LastHopOpinion CAN_HELP*, and the contained node and random partner are locked; else, it is marked *CANT_HELP*.

If a node receives a *PanicSplit* with *LastHopOpinion CAN_HELP*, we perform the same checks with an additional check that we do not hold a lock against the last hop node. If these checks fail, we throw a *PanicSplitFailed* back to the last hop, who unlocks the contained node and partner. We then recurse and treat the message as if *LastHopOpinion* had been *CANT_HELP*. If these checks pass, we proceed as if from a successful *TrySplit* - a *PanicSplitSuccess* is gossiped back to the last hop, our partnership with it is closed and a new connection is initiated with the contained node. The node receiving the *PanicSplitSuccess* message switches connections similarly. The panicking node receiving the connections increases M_c by 1 each, unless it has since stopped panicking, in which case it rejects the connection.

This finds two relatively stable nodes within the network and places the panicking node between them. This is mostly effective in large networks, where the gossiped message will quickly find its way to nodes with no knowledge of the sender. In networks with node count $Nc \leq M + 1$, this message

will never resolve. The resolution rate quickly improves from there.

If $0 < M_c \leq M - 3$, we gossip both of the above messages at once.

If $M_c = 0$, we should assume major disturbance amongst our membership peers. The standard SCAMP leave procedure should be followed to cleanly disconnect from the membership network in the membership manager, before recontacting the origin to gather a fresh partial view and begin new partnerships from scratch.

Each of the above messages contains a time-to-live which is checked at each receiving node. If this timeout expires, the message is dropped. It is the responsibility of each panicking node to resend messages after they time out.

These messages are gossiped to partners, though they could instead be gossiped to members. We suspect that gossiping to partners will bias recipients towards those with a larger number of partnerships (i.e. those with more stable connections) allowing for more effective recovery, whereas the *mCache* may contain a greater concentration of already-poor connections. We cannot provide any proof of this theory, however - this would prove a fruitful future research question.

7.3.4 *Partnerlink* Leaving

To avoid unneeded panics when disconnecting a node, it is intuitive to pair nodes back together, undoing the splitting we performed to enter. If we perform this via messaging, however, failing nodes will still panic all of their nodes, adding M panicking nodes to the network - a substantial overlay breakdown. Instead, we continually inform nodes of their *associated peer* when exchanging buffer maps. A clean leave then commands nodes to switch connections to the node they have already been informed of, and nodes detecting timeouts or a failing node can switch automatically without external assistance. This mechanism allows for overlay repair without any gossiped messages in most cases.

Nodes should be associated deterministically - that is, for as long as a node holds the same partners, the associated peer for each partner should not change. Given a set of partners {A, B, C, ...}, the associated peer for A is B, and for B is A. If there is an odd number of partners, the final peer is associated with an exceptional *NodeID* to be caught later. Peers receiving a buffer map should store the new associated peer corresponding to this partnership, replacing any prior peer. If the partnership fails or times out, the node should first attempt to connect to their associated peer. If this connection fails, or the node was associated with the exceptional *NodeID*, the node should reduce M_c by 1, entering a panic state.

A leaving node should send a *Leave* message to each of its peers before ending the connection. Each receiving node should then immediately attempt the above procedure regardless of the state of the partnership.

7.3.5 *Partnerlink* Switching

coolstreaming-spiked is designed for compatibility with both *DONet* and *New Coolstreaming* research models. To provide this support, we retain the switching system unique to *DONet*. At a set interval, each node should perform a standard *Switch* procedure on a random node in its *mCache*, without changing M_c. Two randomly-chosen associated partners should be locked. If both new links are successful, these two partners should be sent *Leave* messages, so that they reconnect to each other; else, the procedure is abandoned and the two partners are unlocked.

7.3.6 *Partnerlink* Rationale

The *Split* and *Panic* messages are near-identical to those hypothesized to act within the *DONet* model. Switching could also not be achieved without some similar mechanism. These are essential items for the minimum output required of the partner component. The *SplitPanic* system primarily acts as a caution against early failure when connecting to initial candidate partners. One downside of the *Split* model is the chance of collisions - there is some chance that two candidates contact the same node to perform two splits. Network conditions may also lead to a node being impossible to contact. In either case, it is not unreasonable to expect a large number of contacted nodes to be unreachable, resulting in a starting M_c far below M, which would take a long time to recover with the vanilla *Panic* strategy. Whilst performance is not a priority for this model, there is still a minimum bar beyond which research analysis would become clouded by its limitations; starting delay is one such case. The new *associated peers* model is ultimately easier to implement than the *de facto* leave method associated with a splitting approach. Nodes would have to associate peers and forward them to nodes regardless, with the addition of some special case for nodes leaving without notification. The model also grants extra opportunities for filtering . Thus, we consider this approach an improvement over the immediately obvious solution.

discuss limited effectiveness of switching as donet analog

however we phrase this is

7.4 Stream Manager

The stream manager is responsible for converting partnerships to parent-child subscriptions, pushing data around as necessary, as well as the dissemination of buffer maps.

Each node regularly emits a buffer map to every partner. The syntax of these maps has already been discussed - however, an exception on valid syntax is made for when the node has just entered the network. Once buffer maps have been received from some defined minimum percentage of partners, the node must calculate an ideal starting index based on the known set of latest received blocks. Designating $H_{S_i,A}$ as the latest block received block for substream S_i at node A, the starting index at substream i is calculated as

$$I_{S_i} = \max\{H_{S_i,q} : q \in partners\} - T_p$$

where T_p is a constant to be introduced later. The ideal placement of the starting playout index is specific to the playout strategy and is an open research question. We expect that buffering systems should place playout at $\min\{I_{S_i}\}$ and begin playout once a minimum buffer percentage has been filled. Non-buffering approaches are more complicated - playout should be placed such this buffer percentage will have been filled by the time playout reaches $\min\{I_{S_i}\}$. The ideal calculation for this remains an open research question and will likely require additional information to be gossiped between nodes.

When pushing buffer maps to partners when it has not yet received blocks in a given substream i, a new node should fill its latest block with some exceptional value, either a manually-filtered *null* or a very large negative number to ensure the node is never selected, unless the node intends to make its first subscription on i to that partner. In this case, the latest block on this substream should be listed as $I_{S_i} - K$ to ensure the partner's stream manager begins transmission from the relevant point.

When a node n subscribes to a substream through their buffer map, the partnership manager reads and stores the node's latest received block R_n. The manager attends to each subscribing partner with a round-robin strategy. At each step, if the node's buffers contains block with sequence number $R_n + K$, the manager pushes this block to node n and updates R_n accordingly, moving on to the next node. The stream manager should aim to fully saturate a node's outgoing bandwidth - as soon as one block finishes transmission, the next should begin. The stream manager will continue to push all blocks in the substream until the partnership or subscription ends; a parent node will not drop a child under any other circumstance.

For monitoring the service of substream j to child node A by parent p, two inequalities are defined

$$\max\{|H_{S_i,A} - H_{S_j,p}| : i \leq K\} < T_s \quad (1)$$
$$\max\{|H_{S_i,q}| : q \in partners\} - H_{S_j,p} < T_p \quad (2)$$

Inequality (1) caps the distance between the most up-to-date child substream and the target parent substream. If this inequality does not hold, the parent has blocks we are interested in that we are not receiving, implying issues with the link. Inequality (2) caps the distance between the target parent substream and the most up-to-date substream we know across all partners. If this inequality does not hold, the parent is lagging behind the wider swarm, implying connection issues further upstream. At each buffer map and block reception, these inequalities are measured against each substream parent - if either fail, the parent is reselected. The replacement parent is selected randomly from all other partners meeting inequality (2) for the target substream - if no partner meets the inequality, the reselection is cancelled. If any reselection occurs, a new buffer map is immediately sent to the affected parents. A cooldown timer T_i is set on a per-substream basis to prevent repeated reselection and stabilize the overlay topology. Substreams with active cooldowns will not be checked.

The failure state for the stream manager depends on playout strategy. If buffering is incorporated, the node may fall behind the blocks available in the wider swarm. If playout does not halt, we may still find ourselves in a situation where all partners provide unsatisfactory connections. A number of heuristics could detect either case; a threshold over the filled percentage of the buffer is one simple example. Whatever the method, the stream manager should assume a widespread failure amongst its underlying partners, and initiate the same recovery mechanism as defined in the partnership manager.

7.5 Comparison of *coolstreaming-spiked* to our implementation

Our implementation discussed prior is *not* an implementation of *coolstreaming-spiked*, though they are basically similar. The key differences are

- Our implementation retains the *DONet* version of *SCAMP*, with all its oddities. We do not anticipate this aspect of the system to have any impact on performance, and the component outputs are the same.
- Our implementation does not include any locking mechanism - instead, more advanced link counting is used to resolve overlapping message

streams without hiding nodes from overlay adjustment. The interactions that arise are nearly impossible to reason and are very likely the reason for the desyncing M_c and partner counts we note in our results. The locking mechanism is proposed due to our experiences with this other approach. This will have an impact on results, though almost certainly a positive one.

- Our implementation does not make any use of TCP connections, instead relying on UDP either directly or through *OverSim*'s RPC support. This is related both to the cut-down paper we worked from and that TCP support in *OverSim* is a somewhat hidden feature. This negatively impacts our results, as TCP networking is essential in the design of *Coolstreaming*.
- Our implementation initiates playout similarly to a buffering playout strategy, syncing to the starting block index and waiting for a threshold percentage of the buffer to be filled to play. No other buffering is performed during playout. This strange hybrid approach, whilst not invalid, is unlikely to be seen in a real research model and would ideally instead align with one or the other.

The changes we have made to *coolstreaming-spiked* since our implementation aim to resolve the poor performance seen in our results. Whilst the final performance figures are unverified, we believe this to be a good future vein of research.

8 Conclusion

holy fuck. this sucks

References

Beraldi, Roberto, Marco Galiffa, and Hussein Alnuweiri. 2010. W-coolstreaming a protocol for collaborative data streaming for wireless networks. In *2010 ieee 30th international conference on distributed computing systems workshops,* 221–226. https://doi.org/10.1109/ICDCSW.2010.78.

Kondo, Daishi, Yusuke Hirota, Akihiro Fujimoto, Hideki Tode, and Koso Murakami. 2014. P2p live streaming system for multi-view video with fast switching. In *2014 16th international telecommunications network strategy and planning symposium (networks),* 1–7. https://doi.org/10.1109/NETWKS.2014.6959253.

Lan, Shanzhen, Qi Zhang, Xinggong Zhang, and Zongming Guo. 2011. Dynamic asynchronous buffer management to improve data continuity in p2p live streaming. In *2011 3rd international conference on computer research and development,* 2:65–69. https://doi.org/10.1109/ICCRD.2011.5764085.

Li, B., S. Xie, Y. Qu, G. Y. Keung, C. Lin, J. Liu, and X. Zhang. 2008. Inside the new coolstreaming: principles, measurements and performance implications. In *Ieee infocom 2008 - the 27th conference on computer communications,* 1031–1039. https://doi.org/10.1109/INFOCOM.2008.157.

Li, Bo, Susu Xie, Gabriel Y. Keung, Jiangchuan Liu, Ion Stoica, Hui Zhang, and Xinyan Zhang. 2007. An empirical study of the coolstreaming+ system. *IEEE Journal on Selected Areas in Communications* 25 (9): 1627–1639. https://doi.org/10.1109/JSAC.2007.071203.

Xie, Susu, Bo Li, Gabriel Y. Keung, and Xinyan Zhang. 2007. Coolstreaming: design, theory, and practice. *IEEE Transactions on Multimedia* 9 (8): 1661–1671. https://doi.org/10.1109/TMM.2007.907469.

Zhang, Xinyan, Jiangchuan Liu, Bo Li, and Y.-S.P. Yum. 2005. Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. In *Proceedings ieee 24th annual joint conference of the ieee computer and communications societies.* Vol. 3, 2102–2111 vol. 3. https://doi.org/10.1109/INFCOM.2005.1498486.