Joserlin: Joint Request and Service Scheduling for Peer-to-Peer Non-linear Media Access

Zhen Wei Zhao
NUS Graduate School for Integrative Sciences and Engineering
National University of Singapore, Singapore
zhaozhenwei@nus.edu.sg

Wei Tsang Ooi
School of Computing
National University of Singapore, Singapore
ooiwt@comp.nus.edu.sg

ABSTRACT

A peer-to-peer non-linear media streaming system needs to schedule both on-demand and prefetch requests carefully so as to reduce the server load and ensure good user experience. In this work, we propose, Joserlin, a joint request and service scheduling solution that not only alleviates request contentions (requests compete for limited service capacity), but also schedules the prefetch requests by considering their contributions to potential reduction of server load. In particular, we propose a novel request binning algorithm to prevent self-contention among on-demand requests issued from the same peer. A service and rejection policy is devised to resolve contention among on-demand requests issued from different neighbors. More importantly, Joserlin employs a gain function to prioritize prefetch requests at both requesters and responders, and a prefetch request issuing algorithm to fully utilize available upload bandwidth. Evaluation with traces collected from a popular networked virtual environment shows that Joserlin leads to $20\%\sim60\%$ reduction in server load.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Design studies; C.2.4 [**Computer-Communication Networks**]: Distributed Systems - Distributed applications
General Terms: Algorithms, Design, Performance
Keywords: Non-linear access pattern, peer-to-peer, scheduling, service scheduling, on-demand request, prefetch request

1. INTRODUCTION

Peer-to-peer (P2P) techniques have been widely adopted for media streaming. The application of these techniques, however, mainly focuses on linear media, such as video, where user interactions are limited. As new classes of interactive media applications emerge, researchers have started to investigate the use of P2P techniques on streaming these interactive media content, including networked virtual environment (NVE) [9, 18] and zoomable video [21].

User interaction leads to non-linear user access pattern. We say that an access pattern is non-linear if different users may access different sequences of media objects. For instance, in zoomable video, users may choose to zoom into different regions; in an NVE, users may choose to navigate the rooms in a different order. Non-linear access pattern creates *uncertainty* in user accesses: given the current set of media objects accessed, more than one possible set of objects can be accessed next. Such non-linear access pattern poses many new challenges on P2P streaming systems [35], one of which is the request and service scheduling, the focus of this paper.

In a typical P2P media streaming system, peers navigate in the media resource space, consisting of all accessible media objects. Objects not in a peer's cache when they are needed will be requested from neighbors. We call such requests *on-demand requests*. Each on-demand request is associated with a deadline. If the requested object has not been retrieved from a neighbor by the deadline, the request is sent to the server as a last resort. To reduce object retrieval time, prefetching is commonly used. Prefetching predicts objects needed in the future and issue *prefetch requests* to neighbors to retrieve them. Prefetch requests are not sent to the server to reduce the server load.

Our goal in this paper is to schedule on-demand and prefetch requests to reduce the amount of content requested from the server. This goal is equivalent to reducing the number of requests sent to the server, assuming same-size media objects.

We call scheduling activities at the requester side (the peer that sends the request) *request scheduling* and call those at the responder side (the peer that receives the request) *service scheduling*. Request scheduling decides: **(i)** which neighbor a request should be sent to, **(ii)** the requesting order of prefetch requests, and **(iii)** when and at what rate prefetch requests should be sent out. Service scheduling decides: **(i)** which request to serve and which request to reject (referred to as *rejection policy*) and **(ii)** the serving order of requests (referred to as *service policy*).

Request and service scheduling has been studied extensively in the context of P2P VoD streaming systems. In VoD, however, video chunks are accessed mostly sequentially. Seeks are rare (average of $1.6-3.4$ seeks for movies [14] and 9.3 for sports videos [7]), and thus the access pattern is mostly linear. Existing solutions would not work well for non-linear access patterns, due to the following characteristics of interactive media streaming systems:

(i) Data availability changes quickly. Even if a peer A currently holds a media object needed by peer B, the next object needed by B may not be available at A. Existing scheduling solutions that fail to factor in this cannot work well with non-linear access scenarios. For instance, bandwidth reservation-based service scheduling solutions [30, 13, 26] are inefficient, as the reserved bandwidth at a peer A may not be used to serve another peer B if peer A does possess the objects requested by peer B later.

(ii) Prefetch misses frequently. Non-linear access pattern leads to a clear difference between on-demand and prefetch requests – on-demand requests are caused by prefetch misses and have much

MM'13, October 21–25, 2013, Barcelona, Spain.
Copyright 2013 ACM 978-1-4503-2404-5/13/10 ...$15.00.
http://dx.doi.org/10.1145/2502081.2502090.

tighter deadlines. Many existing works do not distinguish the two [26, 29, 18, 28], and thus on-demand requests' deadlines are often missed due to unnecessary contentions with prefetch requests. Solutions that consider the two separately [2] do not study the interplay between the two systematically, particularly how scheduling of prefetch requests can benefit on-demand requests. Existing prefetch request scheduling solutions have been largely heuristic or best-effort, leading to limited benefits to on-demand requests.

To address the aforementioned challenges, we propose Joserlin, a joint request and service scheduling scheme designed for peer-to-peer non-linear media access systems. Joserlin performs request-level scheduling, which is more flexible than the bandwidth reservation-based approaches. Moreover, Joserlin assigns priorities to both on-demand requests (based on urgency) and prefetch requests (based on a derived gain function) to avoid contentions. The derived prefetch gain function factors in prefetch requests' contribution to potential on-demand requests from neighbors that request for the prefetched object, in terms of reducing their probability to be sent to the server. As a result, Joserlin co-schedules on-demand and prefetch requests to reduce the server load due to on-demand requests.

Joserlin encompasses four major contributions: **(i)** We propose a novel request binning algorithm that helps avoid self-contention among on-demand requests from the same peer; **(ii)** We conduct an analytical study on the on-demand request rejection policy; **(iii)** We systematically study the interplay between on-demand and prefetch requests, based on which a gain function is derived to prioritize prefetch requests at both requesters and responders; **(iv)** We propose a prefetch request issuing algorithm that fully utilizes peers' upload bandwidth and respect prefetch requests' relative priority.

The rest of this paper is organized as follows: Sec. 2 discusses related works. Some preliminary preparations are presented in Sec. 3. Sec. 4 investigates on-demand requests. Sec. 5 studies prefetch requests and their interplay with on-demand requests. Evaluation is carried out in Sec. 6 by comparison with existing solutions. Finally, Sec. 7 concludes and presents our future work.

2. RELATED WORK

We now present existing work on request and service scheduling in P2P media streaming, focusing on video and NVE systems.

In P2P live video streaming, *chunk scheduling* (also called *piece selection*) strategies, such as rarest-first [34], random [22], and the hybrid approach [32], are well studied and understood. Guo et al. [11] proposed an optimal scheduling strategy exploiting forward queues. Liang et al. showed that P2P live streaming performance is insensitive to scheduling only when the streaming rate is low or the playback delay is long [17]. Bonald et al. analytically studied the trade-off by a few peer and chunk selection polices in push-based epidemic live streaming systems [6]. They pointed out that the combination of random peer and latest useful chunk selection policy can achieve the optimal dissemination rate within an optimal delay bound. PRIME requests packets from parent peers either randomly or proportionally with respect to their upload bandwidth [20].

For service scheduling, Bertinat et al. [5] proposed a network-flow based service scheduling scheme. Chen et al. [8] proposed a scheduling policy based on Interest-Offer-Accept/Decline. With this approach, responders have to reserve their bandwidth and wait until requesters send back an Accept/Decline message, resulting in long content retrieval latency and a waste of upload bandwidth. These scheduling schemes, however, do not apply to non-linear media access, where different peers access different objects in an asynchronous manner. Furthermore, prefetching, an important consideration in Joserlin, does not exist in live streaming.

In P2P VoD, chunk scheduling strategies such as rarest-first [25, 3] and hybrid approaches [15, 14, 27] are widely adopted by literatures. Apart from that, some works proposed the partition-based method, in which peers' downloading window is partitioned into urgent, normal, and prefetch sections [24]. Peers request data in each section with different probabilities. Kiraly et al. [16] proposed a deadline-based scheduling scheme to prioritize streams of different importances in push-based overlays. Ying et al. [29] assigned requests proportionally to each neighbor's historical throughput.

As for service scheduling in VoD, Zhou et al. [36] investigated strategies such as fair sharing and fixed bandwidth allocation. Yu et al. [30], Huang et al. [13], and Wang et al. [26] proposed their respective service scheduling schemes based on network-flow reservation, which, however, does not fit well with non-linear access scenarios as discussed in Sec. 1. Abbasi et al. [2] used two separate queues: an on-demand queue with earliest deadline first (EDF) service policy and a prefetch queue with FIFO service policy. Their work, however, only provides best-effort scheduling of prefetch requests. Zhang et al. [31] used super nodes to allocate bandwidth, resulting in priority inversion during online scheduling. Zhang et al. [33] proposed a first-aid service scheduling scheme, which allows high priority requests to preempt low priority ones. The work does not deal with prefetch requests and the preempted on-demand requests. Yang et al. [28] suggested sending requests to the least loaded neighbor and using EDF to serve incoming requests. Requests that cannot be served by their deadlines are dropped early. They, however, do not distinguish between on-demand and prefetch requests, leading to contention of the two at the responder side. Furthermore, their do not study the rate at which prefetch requests are issued. These solutions for VoD are not adequate to address the new challenges posed by non-linear accesses, as discussed in Section 1.

Apart from P2P video streaming, several researchers also studied request and service scheduling in P2P NVE streaming. For instance, Chien et al. [9] proposed a reservation-based service scheduling scheme, which is not suitable for non-linear access scenarios (Sec. 1). Moreover, this work does not distinguish on-demand and prefetch requests. Liang et al. [18] used FIFO queue as a service policy. A request that cannot be served before its deadline is rejected and resubmitted to another neighbor until its deadline is reached. The FIFO queue, however, does not factor in the timeliness requirement of requests. Each time a request is resubmitted, it joins the end of the queue, making it even unlikely to be served on time. Another drawback is that the work does not distinguish prefetch requests from on-demand requests. Sung et al. [23] proposed that a peer should send requests to neighbors that are close to its current position with higher probability, and far neighbors with lower probability to alleviate request congestions. By doing this, the method cannot exploit far neighbors that are under-loaded.

Our work differs from existing ones by clearly distinguishing on-demand and prefetch requests, prioritizing them to avoid unnecessary contentions at both requesters and responders, and co-scheduling them to gain from the interplay between the two classes of requests. Moreover, majority of existing solutions treat request and service scheduling as separate, leading to scheduling efficiency loss [28]. We unify the request and service scheduling through a global prioritization policy.

3. PRELIMINARIES

In this section, we characterize a peer's mobility, explain its responsibility as both a requester and responder, and outline assump-

p	a peer
u	the upload bandwidth of a peer
s	the object size, a constant.
q, r	a request (r denotes on-demand request and q denotes prefetch request).
t_r	remaining time until a request r's deadline
C_r	the candidate set of on-demand request r
M_r	the set of neighbors, of the peer that issues request r, possessing the object requested by r, but has not received r. $\lvert M_r \rvert$ is the size.
w_r	the number of remaining retries for request r
ρ	the probability that each serving slot is *unavailable*.
H	random variable for upload bandwidth of a neighboring peer
κ_r	the log-likelihood that request r cannot join any other queue in M_r without rejecting another request.
τ	the prefetch interval size, a constant.
δ_q	the estimated access probability of the object requested by q at the peer that issues q, during the current prefetch interval
ϕ	the access probability threshold to distinguish prefetching gain due to contributions **C1** and **C2**.
B_r	total upload bandwidth of peers in M_r.
g_q	the gain function of a prefetch request q
χ	number of prefetch requests to send to a neighbor at one time, to replenish the queue at that neighbor.

Table 1: Symbol table.

tions made. Table 1 summarizes major symbols used in this paper, and others will be introduced when encountered.

All accessible media objects constitute an n-dimensional media resource space, where peers navigate following their respective paths. We characterize peers' navigation path with (i) its virtual position and (ii) its interest window, centered at the virtual position.

We consider mesh-based P2P protocol, where each peer maintains a list of neighbors. A peer moves around in the resource space to access media objects. A media object that falls within its interest window but is not present in the cache is fetched with an on-demand request, sent to one of its neighbors. Each on-demand request has a timeout value, which is generally short to ensure good user experience. For instance, Claypool et al. [10] reported that the acceptable latency in online interactive games varies from $0.3s$ to $1s$. We model the timeout value as a function of the distance between the requested object and the peer's current virtual position. An on-demand request that cannot be served by the neighbors within its deadline is sent to the server. Furthermore, a peer also prefetches objects from its neighbors.

To serve other peers, each peer maintains a priority queue, where incoming requests are inserted according to a service policy. The peer ensures that all requests in this queue meet their deadlines. Otherwise, one or more requests are rejected from the queue according to a rejection policy. On-demand and prefetch requests may have different service and rejection policies, which will be studied respectively in Sec. 4 and 5. Since on-demand requests have much tighter deadlines, they enjoy higher priority.

Next, we outline assumptions made to simplify the analysis in the remaining of this paper. We assume that media objects are of the same size, denoted by s. Typically, large media objects are often quantized into smaller chunks, each of which can be treated as an independent requestable object. Thus, the same-size object assumption is reasonable. We further assume that each request retrieves one object. In practice, multiple objects can be requested by a single request message.

Joserlin is independent of the access probability prediction algorithm, and we assume that such an algorithm already exists. Moreover, we do not consider incentive mechanism and assume that peers are cooperative. Guo et al. [12] pointed out that some incentive mechanisms such as tit-for-tat is not suitable for streaming systems where peers access media content asynchronously.

4. ON-DEMAND REQUESTS

For on-demand requests, which object to request and when to request are determined by user interactions, rather than by the scheduling scheme. Hence, the on-demand request scheduling scheme only needs to decide: **(Q1)** which neighbor an on-demand request should be sent to? The service scheduling scheme at the responder side needs to decide: **(Q2)** what is the appropriate service and rejection policy?

Since requesters independently issue on-demand requests without consensus among each other, some responders may get overloaded, and others are underloaded. We adopt the reject-and-retry technique used by Liang et al. [18] and Yang et al. [28] to balance the load across neighbors: when an on-demand request is rejected, the requester can quickly retry it to other neighbors until either the request is served or its deadline is reached, by when the request is sent to the server as a last resort. The reject-and-retry technique can effectively balance the load at neighbors, but it does not perform scheduling. Specifically, it does not answer **(Q1)** and **(Q2)**.

Furthermore, on-demand requests often contend for the limited service capacity at responders. Such request contention problem is not addressed by the reject-and-retry technique. For instance, when the system is under heavy load, a portion of the requests have to resort to the server due to capacity deficit at neighbors. With the reject-and-retry technique, however, these requests will be retried multiple times until either timeout or there are no more neighbors to retry, leading to excessive message overhead. Generally speaking, contending requests from the same requester can be avoided at the requester side. Contending requests from different requesters are hard to avoid at responders without introducing significant communication overhead. Therefore, we rely on the request rejection process at the responder side to resolve contention. Joserlin chooses which request to reject carefully, considering both data availability and deadlines of requests.

In the rest of this section, we answer the aforementioned two questions for on-demand requests. More specifically, we propose a request binning algorithm in Sec. 4.1 that decides which neighbor an on-demand request should try, by avoiding self-contention among requests issued from the same requester. Sec. 4.2 studies the service and rejection policy to resolve contentions among requests at the responder side.

4.1 Request Binning Algorithm

Given a set of on-demand requests, the request binning algorithm decides where each request should be sent to. It is designed to alleviate request contention at the requester side, preventing requests issued by the same peer from stepping on each other, i.e., causing a request from itself to be rejected.

We say a set of requests is *contention-free* with respect to a peer p, if all their deadlines can be satisfied by p when there is no other requests at p. The binning algorithm assigns each request in a given set R to either a neighbor or the server. All requests assigned to a neighbor form a *bin* such that requests in the same bin are contention-free with respect to that neighbor. Consider a set of requests. If adding a request r to this set does not cause the deadline of any request (including r) in the set to be violated (with respect to the service policy), we say that this addition is *safe*. We only add a request to a bin if it is safe. Furthermore, for each request $r{\in}R$, the algorithm produces a candidate set C_r. If r is rejected, the requester only retries it to peers in C_r, even if other peers outside C_r may possess the requested object. A neighbor is in C_r if (i) it possesses the object that r is requesting and (ii) adding r to the bin of this neighbor is safe. Alg. 1 sketches the binning algorithm at a generic requester.

The candidate sets allow requests to stop retrying earlier, especially when the system load is heavy. Heavy system load increases the number of requests in each bin, and reduces the size of the candidate sets accordingly. Since we only send rejected requests to neighbors in their respective candidate sets, the number of retries would be small. Our evaluation in Sec. 6 shows that the binning algorithm reduces the retrying message overhead significantly.

Algorithm 1: On-demand Request Binning Algorithm

```
Let $N$ be the set of the requester's neighbors;
Let $R$ be the set of on-demand requests need to be issued;
Let $\lambda_p$ be the total time used by requests at a neighbor $p$;
Let $u_p$ be the upload bandwidth of a neighbor $p$;
Let $\Gamma_p$ be the set of requests (the bin) assigned to a neighbor $p$;
foreach $p{\in}N$ do $\lambda_p{\leftarrow}0$ ;
while $R$ is not empty do
    Remove request $r$ with smallest timeout from $R$;
    Let $N_r$ be a subset of neighbors in $N$ that possess the
    object $o$ requested by $r$;
    Let $t_r$ be the remaining time until $r$'s deadline;
    while $N_r$ is not empty do
        Remove a peer $p$ from $N_r$;
        Let $|o|$ denotes the size of the requested object $o$;
        if $\lambda_p{+}|o|/u_p{\le}t_r$ then
            Add $r$ to $\Gamma_p$;
            $\lambda_p{\leftarrow}\lambda_p{+}|o|/u_p$;
            break;
    if $r$ has not been assigned to any peer then
        Assign $r$ to the server;
Let $C_r$ be the candidate set of a request $r$;
foreach request $r$ that is not assigned to the server do
    foreach $p{\in}N_r$ do
        if $r{\notin}\Gamma_p$ and the insertion of $r$ into $\Gamma_p$ is safe then
            Add $p$ into $C_r$;
```

4.2 Service Policy and Rejection Policy

The service queue of a peer can be modeled as a non-preemptive priority queue, where requests in service cannot be preempted. Since object size is usually small due to chunking of large objects, the non-preemptive priority queue converges to a preemptive priority queue, where the EDF service policy is optimal [19]. We therefore adopt EDF as our service policy for on-demand requests. Note that EDF is also adopted by Yang et al. [28] and Abbasi et al. [2].

We now analyze the rejection policy. Any on-demand request that arrives at peer p and can be safely inserted into the service queue of p (with respect to EDF) will be accepted for service by p. Otherwise, some request r' misses its deadline after the insertion of the newly arrived request as shown in Fig. 1. Let V be the set of requests in the service queue of p that have a earlier or equal deadline than r'. To restore the service queue to a state where each request's deadline is satisfiable, we have to reject one request from V (only one is needed since we assume same-size objects).

We have two naive choices: (i) Reject the newly arrived request, resulting in a drop-tail policy. The drawback of this policy is that, if the newly arrived request's deadline is almost up, it would have insufficient time to be retried at other neighbors and has to be sent to the server. (ii) Reject the request r' whose deadline is violated. Note that r' has the largest remaining time to deadline in V, thus can be retried more times. This choice, however, may cause r' to resort to the server prematurely, if the data availability within its requester's neighborhood is poor (e.g, not many neighbors possess the requested object).

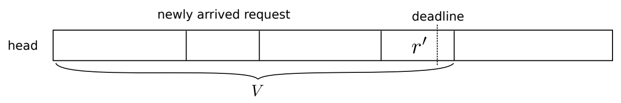


Figure 1: The illustration of V.

Deciding which request to reject needs to factor in both the deadline and the data availability. In general, the rejection policy should reject the request that is most likely to be served from other peers before its deadline. Next, we analyze how these different factors affect the rejection policy.

Consider an on-demand request r that is rejected and needs to be re-sent to another peer. We calculate the log-likelihood that r *cannot* join any other peer's service queue without rejecting a request from them. Let M_r be the neighbors of the requester of r that possess the object requested by r, but has not been tried. Due to the binning algorithm in Sec. 4.1, M_r is a subset of r's candidate set. For tractability purpose, we assume the round-trip time between r's requester and its neighbors is the same, denoted by RTT.

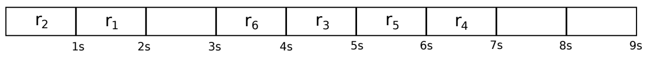


Figure 2: Service queue re-arrangement for on-demand requests. Suppose the original service queue has on-demand requests $\{r_1, r_2, r_3, r_4, r_5, r_6\}$ with remaining time to deadline $\{2s, 2s, 5s, 7s, 7s, 7s\}$ respectively. Rearrangement starts with r_1 being inserted into an available slot that is closest but before its deadline (the second slot in this case). Then r_2 is inserted using the same rule (into the first slot). The rest of requests are inserted in similar fashion.

We denote $x(u, t)$ as the probability that an on-demand request *cannot* join a service queue whose upload bandwidth is u without rejecting a request from that queue, when the request's remaining time to deadline is t. To derive the equation for $x(u, t)$, we divide the time into equal-sized slots. Each slot is the time taken to serve one request. For expository convenience, we re-arrange the service queue in the following manner: iterate through all on-demand requests from the head of the queue, and insert each request into an available slot that is closest to, but before its deadline. We assume that deadlines are always aligned at slot boundaries. A queue rearrangement example is shown in Fig. 2. After the re-arrangement, we can see that $x(u, t)$ is equal to the probability of not finding an available slot before the request's deadline. Let the probability that each slot is *unavailable* be ρ, then

$$x(u, t){=}\rho^{\frac{tu}{s}} \qquad (1)$$

Note that ρ depends on the current system load (the number of on-demand requests). The current system load, however, is hard to characterize as each request has their respective deadlines. Moreover, ρ also depends on the upload capacity, data availability, and peers' access pattern. Modeling ρ is going to be complex, and thus we treat ρ as a constant in this paper.

If the on-demand request r is rejected, the requester randomly selects another neighbor from M_r to retry until r either gets accepted or timeouts. Randomization is necessary to avoid the synchronization effect. The upload bandwidth of the neighbor that r is re-sent to, is therefore a random variable, denoted by H. The number of retries w_r is constrained by $|M_r|$ and the remaining time to deadline t_r:

$$w_r = min\left\{|M_r|, \left\lfloor\frac{t_r}{RTT}\right\rfloor\right\} \tag{2}$$

To calculate the probability that r cannot join any queues in M_r without rejecting another request, consider what needs to happen: r needs to be re-sent to neighbors in M_r, one after another, until r's deadline is up or M_r is exhausted. Every time r is re-sent and rejected, the remaining time t_r reduces by RTT. We can therefore derive the log-likelihood that r cannot join any other queue without rejecting a request as:

$$\begin{aligned}\kappa_r &= E\left[log\left(\prod_{i=1}^{w_r} x(H, t_r - i\times RTT)\right)\right] \\ &= E\left[log\left(\rho^{\frac{H}{s}\sum_{i=1}^{w_r}(t_r - i\times RTT)}\right)\right] \qquad (3)\\ &= \frac{log(\rho)}{s}\times\varphi_r \qquad (4)\end{aligned}$$

where $E[\cdot]$ denotes the expectation and

$$\varphi_r = E[H]w_r\left(t_r - \frac{(w_r+1)RTT}{2}\right) \tag{5}$$

With this analysis, we can now describe the rejection policy. The serving peer should pick the request r with the smallest κ_r to reject. Since $log(\rho)<0$, this selection is equivalent to choosing the request r with the largest φ_r.

$|M_r|$ and $E[H]$ correspond to availability of the object requested by r and t_r corresponds to the remaining time to r's deadline. In general, we should reject requests that have both good data availability and large remaining time. The remaining time, however, is usually the bottleneck if the data availability condition is good. For instance, when $|M_r| \geq \lfloor\frac{t_r}{RTT}\rfloor$, $w_r = \lfloor\frac{t_r}{RTT}\rfloor$ and t_r becomes the dominant factor with a degree of two. Only under poor data availability, factoring in $|M_r|$ and $E[H]$ would help. As the data availability bottleneck is going to be alleviated with prefetch requests in Sec. 5, we find that t_r is usually the bottleneck in the sense that the request r with the largest φ_r is most likely to be the one with the largest remaining time. This observation suggests that the naive rejection policy, which rejects the request whose deadline is violated, would suffice. Therefore, Joserlin uses this as the rejection policy.

5. PREFETCH REQUESTS

Any residual upload bandwidth remaining after serving on-demand requests should be fully utilized to prefetch objects outside the current interest window. On-demand request scheduling only needs to decide which neighbor (or server) an on-demand request should be sent to, as which object to request and when to request are determined by user interactions. Unlike on-demand request scheduling, prefetch request scheduling also needs to decide which object to prefetch and when to prefetch, leading to two additional scheduling dimensions.

As stated in Sec. 3, we assume that there exists a prediction algorithm that assigns each object a probability that the object will be accessed by the peer within a given period of time. Due to user interactions, such prediction algorithms are usually more accurate when this time period is short. Therefore, Joserlin slots time into small prefetch intervals of length τ (typically in the order of seconds) and each peer recomputes objects' access probability at the beginning of each prefetch interval. Each prefetch request therefore has a timeout value of τ. Unlike on-demand request, the deadline for a prefetch request exists to ensure that: (i) the prefetch request is not obsolete due to recomputation of access probability after every prefetch interval, and (ii) the requester gets a chance to re-send the request to another neighbor if the current neighbor is overloaded. Note that, for a prefetch request, it is rejected if the serving peer cannot *start* serving it by the deadline, while for an on-demand request, it is rejected if the serving peer cannot *finish* serving it by the deadline. This minor modification is to prevent scenarios where the remaining time to the next prefetch interval cannot fit in a prefetch request, leading to wasted bandwidth.

With the preceding prefetching framework, there are still two questions unanswered: **(Q3)** How to decide the requesting and serving order of prefetch requests? **(Q4)** When and at what rate should prefetch requests be issued so as to fully utilize the available upload bandwidth? To answer **(Q3)**, Sec. 5.1 derives a gain function, which is used to prioritize prefetch requests at both requesters and responders. In Sec. 5.2, a prefetch request issuing algorithm is devised to answer **(Q4)**.

5.1 Prefetch Gain Function

We associate each prefetch request q with a gain function g_q to prioritize them for both issuing and serving order. This gain function reflects the contributions of q to server load reduction. Higher gain leads to higher priority.

In general, a prefetched object contributes to reducing the server load in three ways: **(C1):** it eliminates on-demand requests for the same object (if the prefetch is a hit) at the prefetching peer. **(C2):** it increases data availability and, therefore, increases the chances that a neighbor of the prefetching peer can successfully fetch the object via on-demand requests. **(C3):** it increases data availability and, therefore, increases the chances that a neighbor of the prefetching peer can prefetch the object via prefetch requests. **C3** can be extended to multiple hops.

Both **C2** and **C3** help to reduce the number of on-demand requests sent by neighbors to the server: **C2** allows a neighbor's on-demand requests to be met, and **C3** suppresses an on-demand request in case the prefetch request for the same object hits. However, we do not consider **C3** in our gain function, since it is hard to tell if a missing object will be requested on-demand or prefetched by a neighbor. But, in the worst case, if prefetch requests fail to retrieve the object and the object is needed, an on-demand request will be issued by the neighbor anyway. It is therefore more important to consider **C2**. Henceforth, we will only consider **C1** and **C2** when designing the gain function.

Among **C1** and **C2**, we give priority to **C1** over **C2**, since if we can eliminate on-demand requests in the first place, there is no need for the second type of contribution. Furthermore, prefetch hit leads to better user experience compared to on-demand requests.

Consider a prefetch request q. Let δ_q be the access probability of the object requested by q at the peer that issues q within the current prefetch interval, as estimated by the prediction algorithm. If δ_q is larger than a threshold ϕ, we treat the corresponding prefetch request q as having significant gain due to **C1**. Otherwise, its gain due to **C1** is negligible, and we should consider its contribution

C2. If $\delta_q > \phi$, the contribution of q can be modeled using the access probability δ_q:

$$g_1(q) = \delta_q \tag{6}$$

If $\delta_q \leq \phi$, the gain due to **C2** is not obvious, and its derivation requires a comprehensive analysis of the effect that this prefetch request may potentially have on a neighbor's on-demand request. In Eqn. 4, we have derived the log-likelihood that an on-demand request r cannot be served by any neighbor without rejecting an existing on-demand request in their service queues. We can regard κ_r as a function of $|M_r|$ and $B_r = E[H]|M_r|$, where B_r is the total upload bandwidth of peers in M_r.

$$\kappa_r(|M_r|, B_r) = \frac{log(\rho)B_r}{s|M_r|} w_r \left(t_r - \frac{(w_r+1)RTT}{2} \right)$$

Note that when r have not been sent to any peer, the remaining time to deadline t_r is equal to its timeout value.

Now consider what would happen when one neighbor of the prefetching peer sends an on-demand request r for the prefetched object: M_r is going to expand with the prefetching peer by 1, and B_r is going to increase by u, which is the upload bandwidth of the prefetching peer. As a result, r will have a higher chance of being served. We can quantify this increment with the decrease in log-likelihood

$$\kappa_r(|M_r|, B_r) - \kappa_r(|M_r|+1, B_r+u) = -\frac{log(\rho)}{s} \times \theta_r^u \tag{7}$$

where

$$\theta_r^u = \begin{cases} t_r u - \frac{RTT}{2}(B_r + |M_r|u + u) & \text{if } |M_r| < \lfloor \frac{t_r}{RTT} \rfloor \\ \frac{t_r^2}{2RTT} \times \frac{|M_r|u - B_r}{|M_r|(|M_r|+1)} & \text{otherwise} \end{cases} \tag{8}$$

Note that we ignore some boundary conditions for the sake of simplicity in Eqn. 8. As the term $-log(\rho)/s$ is positive and is a constant, we factor it out and use θ_r^u as the gain function for prefetching the object that would be needed by a neighbor's on-demand request r.

Note that θ_r^u could be negative – i.e., prefetching the object actually "hurts" the neighbors' on-demand requests. This scenario can occur, for instance, when the prefetching peer's upload bandwidth u is very small. A peer with small upload bandwidth may not help much in serving on-demand requests, but may cause a peer to waste one RTT, increasing the chance of timeout. We, however, want to bound the gain to non-negative values only. This is reasonable, since after a peer prefetches an object, it can opt not to announce itself as the content provider for that object (e.g., if its upload bandwidth is too small).

We can now derive the gain of a prefetch request q due to **C2**. Let u be the upload bandwidth of the prefetching peer that issues q, and let N_q be neighbors of the prefetching peer that do *not* possess the object requested by q. If a neighbor already possesses an object, it would not issue any on-demand request for it. Consider each peer $p \in N_q$ and their corresponding on-demand request r_p for the same object requested by q. The gain function can be written as:

$$g_2(q) = \sum_{p \in N_q} max(0, \theta_{r_p}^u). \tag{9}$$

To compute g_2, the prefetching peer reports its upload bandwidth to each of its neighbors p. For each object that is missing from p's cache, p collects the number of content providers for the object in its neighborhood, the total upload bandwidth of the content providers, and the average RTT between itself and these content providers. The timeout value of an on-demand request is only known when it is issued. The average timeout value of historical on-demand requests, however, can be used as approximation. With these information, p computes the gain of the prefetch request q to the on-demand request that is issued from itself and requests for the same object using Eqn. 8, and then sends it back to the prefetching peer. The prefetching peer aggregates potential gains from all of its neighbors and computes g_2 using Eqn. 9.

Finally, we combine the two contributions and yield the prefetch gain function for a prefetch request q as

$$g_q = \begin{cases} g_1(q) & \text{if } \delta_q > \phi \\ \frac{g_2(q)}{G_{max}} \times \phi & \text{otherwise} \end{cases} \tag{10}$$

G_{max} is a constant, denoting the maximum possible value of g_2. The purpose of introducing this constant is to normalize g_2 to the range of [0, 1].

After receiving a prefetch request, the serving peer inserts it into the same service queue as on-demand requests. On-demand requests have higher priority over prefetch requests in the service queue. Within on-demand requests, priority is given to the one with earlier deadline, and within prefetch requests, priority is given to the one with larger gain value. Ties are broken based on the objects' distance to the prefetching peers' current virtual positions.

5.2 Prefetch Request Issuing Algorithm

We now discuss Joserlin's prefetch request issuing algorithm that decides when and where a prefetch request should be sent to. The algorithm aims to: (i) fully utilize peers' upload bandwidth, (ii) respect the relative priority of prefetch requests issued from different peers, and (iii) incur small overhead.

Fully utilizing peers' upload bandwidth is not difficult if peers issue prefetch requests quickly. This behavior, however, would create serious problem if prefetch requests are not prioritized properly at the responder side. For instance, if prefetch requests are served using the FIFO policy [2], aggressively issuing prefetch requests would starve other peers. In contrast, if prefetch requests are issued conservatively, peers' upload bandwidth may be under utilized. In Joserlin, since peers serve prefetch requests in decreasing order of priority (based on the gain function in Sec. 5.1), how fast a neighbor's prefetch requests get served depends on how much gain they bring to the system compared to prefetch requests from other neighbors, not how fast they arrive. Thus, there is no point in issuing prefetch requests aggressively. A peer in Joserlin therefore should issue prefetch requests at a rate that matches the serving rate at each of its neighbors.

Noticing that peers' upload bandwidth gets fully utilized as long as their service queues are not empty, our idea is that each peer should ensure that it has at least one prefetch request awaiting in each of its neighbors' service queue *at any point of time* (except special cases, such as when the peer has no prefetch requests to issue). Ensuring this property also allows the serving peer to respect the "global" priority, i.e., serve prefetch requests in decreasing order of priority *as computed across multiple prefetching peers*.

Note that a peer with upload bandwidth of u can serve up to $\varpi = \lceil \frac{u \times RTT}{s} \rceil$ number of prefetch requests from a neighbor within an RTT. To retain the system at a valid state, a neighbor should issue new prefetch requests to the service queue, whenever it realizes the number of its pending prefetch requests at the service queue drops below ϖ.

Based on the aforementioned observations, we design Joserlin's prefetch request issuing algorithm as follows: at the beginning of each prefetch interval, a prefetching peer builds a priority queue of all prefetch requests it will send, using the gain function as the priority. Then, for each of the peer's neighbors (in random order), the peer picks up to $\varpi{+}\chi$ ($\chi{\geq}1$) highest priority prefetch requests for objects available at that neighbor, and send these requests to it. For every χ prefetched objects received from a neighbor, the prefetching peer issues another χ prefetch requests to this neighbor. χ is the replenishing unit. Rejected prefetch requests are re-inserted into the priority queue. The prefetching peer stops sending prefetch requests to a neighbor that rejects a prefetch request until the next prefetch interval. In practice, χ prefetch requests can be bundled into a single message. Small χ causes more request message overhead. Large χ, however, causes more rejection message overhead. Joserlin uses $\chi{=}\varpi$ to adapt to RTT and upload bandwidth of the neighbors.

6. EVALUATION

In this section, we evaluate Joserlin with traces collected from a well-known NVE called Second Life and a simulator implemented on top of Oversim [4]. We choose NVE as the application to base our evaluation on because (i) it is a typical non-linear access scenario; (ii) there are many existing work on peer-to-peer NVE, allowing us to compare our methods; and (iii) user traces are available and not difficult to collect.

6.1 Trace Collection and Parameter Settings

The traces include both avatar mobility traces(we do not distinguish avatar and peer in the rest of this paper) and texture traces collected from the Second Life region Freebies and Sunland. The size of both regions is $256m{\times}256m$. The mobility traces sample avatar's position every $10s$, and we interpolate the positions in between to obtain traces that sample every $1s$. The Freebies trace is 2.5 hours long and the Sunland trace is 2.2 hours long. The media objects that avatars need to retrieve while moving around are textures. There are 1782 textures in the Freebies region and 2197 in the Sunland region. Each texture has a position and size value. Large textures are quantized into units of 64KB. After quantization, the number of texture objects becomes 3422 and 5747, respectively.

We use the same bandwidth trace as the one used by Liang et al. [18], which is collected from dslreports.com [1]. The startup latency is set to 5s. Each peer maintains between 10 and 15 neighbors. If the number of neighbors drops below 10, the peer should request new neighbors from a tracker. Both bitmap exchange and gain update intervals are set to 10s.

The interest window in our model corresponds to the Area-of-Interest in NVE, which we set to a circle with a radius of 20m (the interest window size, ψ). The deadline to retrieve a texture is set to be a linear function of d, the distance between the texture and the peer's current position:

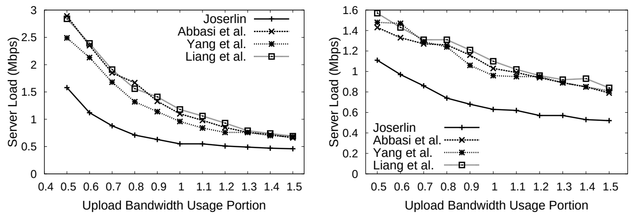
$$f(d){=}t_{min}{+}\frac{d}{\psi}(t_{max}{-}t_{min}) \qquad (11)$$

where t_{min} and t_{max} are the minimum and maximum timeout value for on-demand requests. By default, we set t_{min} to 0.3s and t_{max} to 1s.

The prefetch interval size τ is set to 5s. To predict which media object is needed (for prefetching), we assume that a peer continues to move in the same direction and with the same speed for the next τ seconds. The predicted distance moved is thus τ times the current moving speed. A cap of 30m is put on the predicted distance moved to prevent exceedingly large instantaneous moving speed, caused by user interactions such as teleportation. Based on the predicted movement, a predicted interest region can be computed. Textures that fall within the predicted region are considered to have predicted access probabilities larger than the threshold ϕ. All other textures have access probabilities less than ϕ.

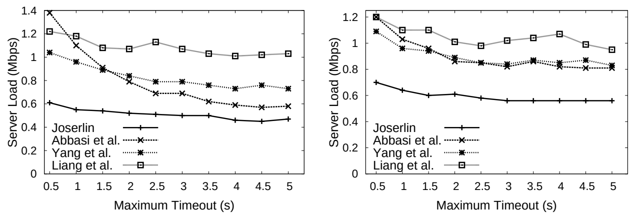
Within the predicted region, textures that are further away from the peer's current position have smaller access probabilities. To model this, we set a texture's access probabilities to $\frac{d}{D_{max}}(\phi{-}1){+}1$, where d denotes the texture's distance from the peer's current position, and D_{max} is the maximum possible value of d (used to normalize d to the range of $[0, 1]$). This function restricts the access probability of any texture within the predicted region to be in the range of $[\phi, 1]$. Note that the value for ϕ does not matter for our evaluation, as it is not necessary to know the exact predicted access probability. Knowing the relative access probability is sufficient to break ties among different prefetch requests. Furthermore, note that we need not model the access probability for textures outside of the predicted region (Eqn. 10).

In our simulations, serving peers serve one request at a time. Although in practice, peers may serve several requests concurrently. Furthermore, network layer details such as congestion and packet loss are not captured in the simulations.



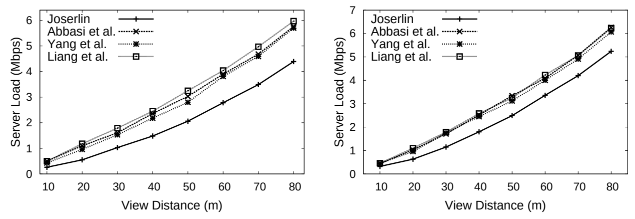
(a) Freebies (b) Sunland

Figure 3: Vary the upload bandwidth.



(a) Freebies (b) Sunland

Figure 4: Vary the timeout value.



(a) Freebies (b) Sunland

Figure 5: Vary the interest window size.

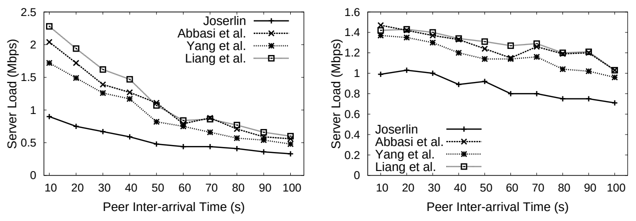
6.2 Performance Comparison

We compare our request and service scheduling scheme with solutions proposed by Liang et al. [18], Yang et al. [28], and Abbasi et al. [2]. We use the same parameter settings for all four schemes, including the prefetch prediction algorithm.

Abbasi et al. [2] use cooperative prefetching, with which textures inside the predicted region are prioritized over those outside. Ties among objects that fall into the predicted region are broken based on their distance to peers' current positions. The rarest-first policy is used for objects outside the region. Liang et al. [18]'s and Yang

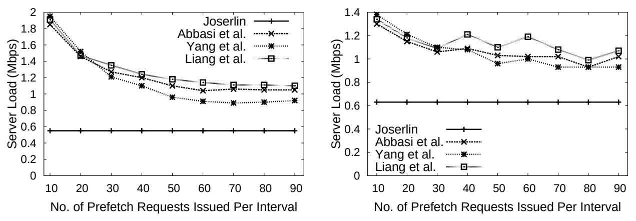
		default settings		50% of upload band.		timeout range [0.3, 2]		40m view distance		30s inter-arrival time	
		timeout	data avail.	timeout	data avail.	timeout	data avail.	timeout	data avail.	timeout	data avail.
Freebies	Yang et al.	614	20,023	936	52,474	174	17,411	825	44,922	444	26,270
	Liang et al.	1,026	25,479	1,676	61,277	547	23,426	1,857	52,947	1,239	34,961
	Abbasi et al.	318	22,511	691	57,829	44	16,540	891	47,557	345	27,781
	Joserlin	203	12,242	1,106	31,587	95	11,760	379	31,768	496	14,292
Sunland	Yang et al.	744	18,178	1,265	28,289	403	17,008	1,580	45,694	229	25,209
	Liang et al.	1,189	21,190	753	31,450	912	19,640	2,399	49,873	279	28,047
	Abbasi et al.	920	19,031	296	27,696	348	16,412	1,454	46,926	641	25,622
	Joserlin	362	12,123	751	20,866	115	11,946	396	34,818	207	18,728

Table 2: Number of requests sent to the server and their respective reasons.



(a) Freebies (b) Sunland

Figure 6: Vary the peer arrival rate.

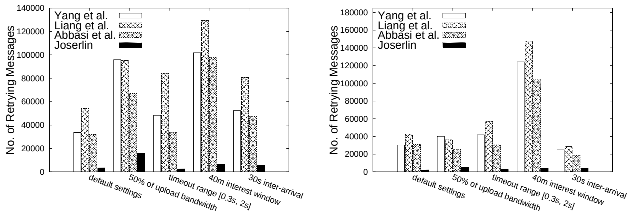


(a) Freebies (b) Sunland

Figure 7: Vary the number of prefetched objects per prefetch interval.

et al. [28]'s solutions do not prioritize prefetch requests, so we augment them with one: Textures in the predicted region are given higher priority. Within each category (textures within or outside the predicted region), closer textures get higher priority.

The three schemes that we are comparing against do not have a complete prefetch request issuing algorithm. Yang et al. [28] proposed that each peer can send n requests to each of its neighbors periodically, but did not hint on how large n should be. To ensure a fair comparison, for these three schemes, each peer sends 50 prefetch requests to its neighbors at the beginning of each prefetch interval. Ignoring the data availability bottleneck, 50 prefetch requests is enough to fully utilize the available upload bandwidth. We vary this number as well in our performance comparison to study its effect on these schemes. Note that the three schemes have their respective service policy for prefetch requests: Liang et al. [18] and Abbasi et al. [2] use FIFO policy; Yang et al. [28] use EDF policy. Moreover, we augment Abbasi et al.'s scheme with reject-and-retry, even though this technique is not in their proposed solution.



(a) Freebies (b) Sunland

Figure 8: Number of retrying messages.

Upload Bandwidth. In Fig. 3, we vary peers' upload bandwidth from 50% to 150% of their default values and plot the average server load with respect to the upload bandwidth. As expected, the server load decreases as the upload bandwidth increases. Joserlin achieves a performance gain of 30%∼50% for the Freebies trace and 25%∼40% for the Sunland trace, in comparison to the other three schemes. In Fig. 3(a), as the upload bandwidth decreases, the server load of the other three schemes increases much faster than Joserlin's. This observation suggests that Joserlin is particularly good at handling relatively heavy system load (with respect to the available upload bandwidth). As the upload bandwidth increases, the four schemes converge in Fig. 3(a). This is because the system performance will be insensitive to scheduling when there is excessive upload bandwidth available. In Fig. 3(b), such trend is less obvious, as the user access pattern in the Sunland trace is bursty, leading to higher instantaneous system load.

Timeout. Fig. 4 keeps the t_{min} value unchanged and varies the maximum timeout value t_{max}. Joserlin achieves a performance gain of 35%∼40% over Yang et al.'s scheme, 50%∼55% over Liang et al.'s scheme, 55% over Abbasi et al.'s scheme when timeout is small and 20% when timeout is large for the Freebies trace. As for the Sunland trace, the performance gain is 30%∼35% over Yang et al.'s scheme, 40%∼47% over Liang et al.'s scheme, 40% over Abbasi et al.'s scheme when timeout is small and 30% when timeout is large.

It is worth noticing that Joserlin's server load only increases slightly as t_{max} decreases. The server load for each of the other three schemes, however, either remains high or increases quickly when t_{max} decreases. In general, the significance of prefetch requests to on-demand requests is much larger when the timeout value is small. With our carefully designed prefetch gain function and issuing algorithm, Joserlin successfully exploits the interplay between prefetch and on-demand requests, thus incurs much less server load when the timeout value is small.

In Fig. 4(a), the performance of Yang et al.'s and Liang et al.'s schemes does not change much with respect to the timeout value, since they do not differentiate on-demand and prefetch requests, causing contentions between the two. Abbasi et al.'s scheme incurs a large server load when the timeout value is small due to their simple prefetch scheduling policy. As the timeout value increases, Abbasi et al.'s scheme, which differentiates on-demand and prefetch requests, beats the other two and converges to Joserlin's performance. Most media streaming systems, however, demand short timeout value to ensure good user experience.

Interest Window Size. The interest window size is varied in Fig. 5. As the interest windows size increases, instantaneous system load increases as the more objects need to be fetched on-demand when prefetch misses occur. The performance gain of Joserlin over the other three schemes decreases from 40% to 20% for the Freebies trace, and decreases from 35% to 15% for the Sunland trace, as the interest window size increases. When the instantaneous system load is high, scheduling becomes insignificant, as there is insufficient upload capacity to serve the requests anyway. Note that there is no contradiction with the observation in Fig. 3(a), where Joserlin can handle relatively heavy system load better than other schemes. When the system load is light, we do not need sophisticated scheduling, and when the system load is too heavy, scheduling becomes insignificant. The scheduling scheme makes a difference only when the system load is somewhere in between.

Peer Arrival Rate. We separate the two traces into one mobility sequence per peer, and each sequence is fed into the simulator following a Poisson peer arrival rate with different average inter-arrival time, until they are drained. In Fig. 6(a), Joserlin achieves a performance gain of $50\%\sim60\%$ compared to the other schemes when the peer arrival rate is large. When peer arrival rate is small, the performance gain decreases to $30\%\sim45\%$. This observation suggests that Joserlin is good at dealing with large peer populations. In Fig. 6(b), the performance gain is $20\%\sim30\%$.

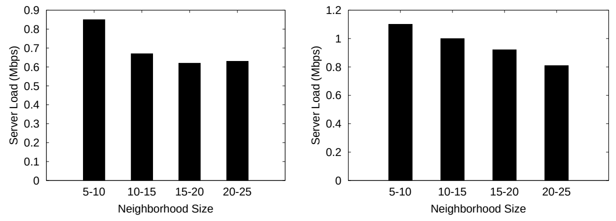
Number of Prefetch Requests. In Fig. 7, we vary the number of prefetch requests issued by Yang et al's, Liang et al.'s and Abbasi et al.'s scheme per prefetch interval. Note that this parameter is not used in Joserlin's prefetch request issuing algorithm. Instead, Joserlin automatically adopts its prefetch request issuing rate to the serving rate. Joserlin, whose results are shown as a horizontal lines in the figure, beats the other three schemes when we vary the aggressiveness of prefetching. We can see that the server load suffers when fewer prefetch requests are issued per interval, due to under utilization of peers' upload bandwidth. As more prefetch requests are issued, the server load, however, stabilizes. These two figures indicate that simply sending more prefetch requests to saturate peers' upload bandwidth is not enough. Sophisticated scheduling for prefetch requests is needed.

Causes of Server Load. Next, we select five representative scenarios to investigate the causes of server load and the retrying message overhead. These five representative scenarios are the default setting, 50% of upload bandwidth, timeout range of [0.3, 2], $40m$ viewing distance, and $30s$ peer inter-arrival time.

In Table 2, we inspect the causes of server load by investigating why each request is sent to the server in the five representative scenarios. The reason is either "timeout", if the request's deadline is reached, or "data availability", if the neighbors either do not possess the requested object or have insufficient upload capacity to serve the request. Joserlin significantly reduces the number of on-demand requests sent to the server due to "data availability", which is the sole major cause of the server load. The prefetch gain function in Sec. 5.1 and prefetch request issuing algorithm in Sec. 5.2 contribute to this reduction.

Number of Retries. We compare the number of retries in Fig. 8. Our scheme incurs significantly fewer retrying messages under all scenarios. In particular, comparing the scenario with timeout range of [0.3s, 2s] to the default setting, the retrying message overhead for both Yang et al.'s and Liang et al.'s schemes increases due to request contentions. The larger timeout value causes requests to retry more times before timeout. When larger interest window size is used, all other three schemes incur excessive retrying message overhead due to the heavier system load. The reject-and-retry mechanism retries many requests repeatedly until timeout, as there is insufficient service capacity. Joserlin avoids this problem with the on-demand request binning algorithm sketched in Alg. 1, thus incurs much less overhead. For instance, in the Freebies trace with timeout range of $[0.3s,\ 2s]$ scenario, Joserlin's retrying message overhead is approximately 7.5%, 5.3%, and 3% of that incurred by Abbasi et al.'s, Yang et al.'s, and Liang et al.'s respectively.

Neighborhood Sizes. We further use the Freebies trace with $30s$ inter-arrival time scenario to inspect the effect of neighborhood



(a) Freebies (b) Sunland

Figure 9: Effect of different neighborhood sizes.

size. The server load is compared between neighborhood sizes distributed between $5-10$, $10-15$, $15-20$, and $20-25$. In Fig. 9(a), the server load incurred by neighborhood size of $10-15$, $15-20$, and $20-25$ are very similar. In Fig. 9(b), however, the server load decreases as the neighborhood size increases. We believes that this phenomenon is due to the higher instantaneous system load in the Sunland trace, and thus more neighbors are needed to absorb the burst of on-demand requests. A general guideline for the neighborhood size would be to use large neighborhood size if the instantaneous system load is higher. Otherwise, small neighborhood size would suffice, helping reduce the message overhead for neighborhood maintenance and bitmap exchange.

Protocol Overhead. The protocol overhead is mainly caused by the underlying distributed computation of gain, which requires exchanges of θ. The interval for recomputation of the gain function is set to $10s$. Thus, if the neighborhood size is between $10-15$, each peer needs to send $1-1.5$ update messages per second. By our measurement, the average value is 1.23/s. Each update message should include the computed θ value (Eqn. 8) for missing objects. Note, however, that a peer does not need to report the θ value to its neighbor under the following circumstances: (i) the object is present in its cache or has been requested, (ii) the θ value does not change since last report, and (iii) its neighbor already has the object in its cache. This observation allows us to significantly prune the number of gain values reported in each update message. Further, each gain value can be encoded with $2-4$ bytes depending on the quantization and accuracy. In our simulations, we measured 26.18 values per message on average for the Freebies trace and 22.61 for the Sunland trace, using default settings.

7. CONCLUSION AND FUTURE WORK

To conclude, Joserlin is a joint request and service scheduling scheme for P2P non-linear media streaming systems that avoids request contention through request binning and exploits the interplay between on-demand and prefetch requests to prioritize prefetch requests at both requesters and responders. Joserlin issues prefetch requests to fully utilize peers' upload bandwidth, without violating requests' relative priorities. Our evaluation compares Joserlin with three existing solutions using traces collected from Second Life and finds that Joserlin outperforms these solutions in all scenarios.

For future work, we would like to investigate how to maintain neighborhood in the P2P overlay to further improve streaming efficiency and reduce the server load. If neighbors of a powerful peer have prefetched enough content, we should restructure the overlay so that this peer can become a neighbor of other more needy peers. Such restructuring adds a new optimization dimension to our joint request and service scheduling.

Acknowledgments: This research is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.

8. REFERENCES

[1] Broadband reports and speed test statistics. http://www.dslreports.com/archive.

[2] U. Abbasi, G. Simo, and T. Ahmed. Differentiated chunk scheduling for P2P video-on-demand system. In *CCNC'11*, pages 622–626, Las Vegas, NV, January 2011.

[3] S. Annapureddy, S. Guha, C. Gkantsidis, D. Gunawardena, and P. R. Rodriguez. Is high-quality VoD feasible using P2P swarming? In *WWW'07*, pages 903–912, Banff, Canada, May 2007.

[4] I. Baumgart, B. Heep, and S. Krause. OverSim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84. IEEE, 2007.

[5] M. E. Bertinat, D. Padula, F. R. Amoza, P. Rodríguez-Bocca, and P. Romero. Optimal bandwidth allocation in mesh-based peer-to-peer streaming networks. In *INOC'11*, pages 529–534, Hamburg, Germany, June 2011.

[6] T. Bonald, L. Massoulié, F. Mathieu, D. Perino, and A. Twigg. Epidemic live streaming: optimal performance trade-offs. *SIGMETRICS Perform. Eval. Rev.*, 36(1):325–336, June 2008.

[7] A. Brampton, A. MacQuire, I. A. Rai, N. J. P. Race, L. Mathy, and M. Fry. Characterising user interactivity for sports video-on-demand. In *NOSSDAV'07*, pages 99–104, Urbana-Champaign, IL, June 2007.

[8] Z. Chen, K. Xue, and P. Hong. A study on reducing chunk scheduling delay for mesh-based P2P live streaming. In *Grid and Cooperative Computing (GCC'08)*, pages 356–361, Shenzhen, China, October 2008.

[9] C.-H. Chien, S.-Y. Hu, J.-R. Jiang, and C.-W. Cheng. Bandwidth-aware peer-to-peer 3D streaming. *Int. J. Adv. Media Commun.*, 4(4):324–342, Nov. 2010.

[10] M. Claypool and K. Claypool. Latency and player actions in online games. *Commun. ACM*, 49:40–45, November 2006.

[11] Y. Guo, C. Liang, and Y. Liu. AQCS: adaptive queue-based chunk scheduling for P2P live streaming. In *IFIP-TC6 Networking Conference (NETWORKING'08)*, pages 433–444, Singapore, May 2008.

[12] Y. Guo, S. Mathur, K. Ramaswamy, S. Yu, and B. Patel. Ponder: Performance aware P2P video-on-demand service. In *GLOBECOM'07*, pages 225–230, Washington, DC, Nov. 2007.

[13] C. Huang, J. Li, and K. W. Ross. Peer-assisted VoD: Making Internet video distribution cheap. In *IPTPS'07*, Bellevue, WA, Feb. 2007.

[14] Y. Huang, T. Z. J. Fu, D. M. Chiu, J. C. S. Lui, and C. Huang. Challenges, design and analysis of a large-scale P2P VoD system. In *SIGCOMM'08*, pages 375–388, Seattle, WA, Aug. 2008.

[15] K.-W. Hwang, V. Misra, and D. S. Rubenstein. Stored media streaming in BitTorrent-like P2P networks. Technical Report CUCS-024-08, Columbia University, New York, April 2008.

[16] C. Király, R. L. Cigno, and L. Abeni. Deadline-based differentiation in P2P streaming. In *GLOBECOM'10*, pages 1–6, Miami, FL, Dec. 2010.

[17] C. Liang, Y. Guo, and Y. Liu. Is random scheduling sufficient in P2P video streaming? In *ICDCS'08*, pages 53–60, Beijing, China, June 2008.

[18] K. Liang, R. Zimmermann, and W. T. Ooi. Peer-assisted texture streaming in metaverses. In *MM'11*, pages 203–212, Scottsdale, AZ, Nov. 2011.

[19] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[20] N. Magharei and R. Rejaie. PRIME: peer-to-peer receiver-driven mesh-based streaming. *IEEE/ACM Trans. Netw.*, 17(4):1052–1065, Aug. 2009.

[21] A. Mavlankar, J. Noh, P. Baccichet, and B. Girod. Peer-to-peer multicast live video streaming with interactive virtual pan/tilt/zoom functionality. In *ICIP'08*, pages 2296–2299. IEEE, 2008.

[22] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *IPTPS'05*, pages 127–140, Ithaca, NY, Feb. 2005.

[23] W.-L. Sung, S.-Y. Hu, and J.-R. Jiang. Selection strategies for peer-to-peer 3D streaming. In *NOSSDAV'08*, pages 15–20, Braunschweig, Germany, May 2008.

[24] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for supporting streaming applications. In *INFOCOM'06*, pages 1–6, Barcelona, Spain, Apr. 2006.

[25] N. Vratonjić, P. Gupta, N. Knežević, D. Kostić, and A. Rowstron. Enabling DVD-like features in P2P video-on-demand systems. In *P2P-TV'07*, pages 329–334, Kyoto, Japan, Aug. 2007.

[26] J. Wang, C. Huang, and J. Li. On ISP-friendly rate allocation for peer-assisted VoD. In *MM'08*, pages 279–288, Seattle, WA, Aug. 2008.

[27] X. Yang, M. Gjoka, P. Chhabra, A. Markopoulou, and P. Rodriguez. Kangaroo: Video seeking in P2P systems. In *IPTPS'09*, Boston, MA, Apr. 2009.

[28] Y. Yang, A. L. H.Chow, L. Golubchik, and D. Bragg. Improving QoS in BitTorrent-like VoD systems. In *INFOCOM'10*, pages 2061–2069, San Diego, CA, Mar. 2010.

[29] L. Ying and A. Basu. pcVOD: Internet peer-to-peer video-on-demand with storage caching on peers. In *International Conference on Distributed Multimedia Systems (DMS'07)*, pages 218–223, San Francisco, CA, Sept. 2007.

[30] Q. Yu and D. Chen. Optimal data scheduling for P2P VoD streaming systems. In *ICPADS'10*, pages 817–822, Shanghai, China, Dec. 2010.

[31] M. Zhang and B. Feng. A P2P VoD system using dynamic priority. In *Malaysia International Conference on Communication*, pages 518 – 523, Kuala Lumpur, Malaysia, Dec. 2009.

[32] M. Zhang, Y. Xiong, Q. Zhang, and S. Yang. On the optimal scheduling for media streaming in data-driven overlay networks. In *GLOBECOM'06*, San Francisco, CA, Nov. 2006.

[33] T. Zhang, F. Ren, and X. Cheng. First-aid mechanism for peer-to-peer streaming. In *ICNP'08*, Orlando, FL, Oct. 2008.

[34] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. CoolStreaming/DONet: A data-driven overlay network for peer-to-peer live media streaming. In *INFOCOM'05*, pages 2102–2111, Miami, FL, Mar. 2005.

[35] Z. W. Zhao. Challenges in supporting non-linear and non-continuous media access in P2P systems. In *MM'12*, pages 1397–1400, Nara, Japan, Nov. 2012.

[36] Y. Zhou, T. Z. J. Fu, and D. M. Chiu. A unifying model and analysis of P2P VoD replication and scheduling. In *INFOCOM'12*, pages 1530–1538, Orlando, FL, Mar. 2012.