Trading off resources between overlapping overlays

Brian F. Cooper

College of Computing, Georgia Institute of Technology
cooperb@cc.gatech.edu

Abstract. Many different overlays with different properties have been proposed. Rather than using one overlay for all applications, it is likely that multiple overlapping overlays will be deployed on the same computing resources for different purposes. We present an architecture, called ODIN-S, for mediating the resources used by overlapping overlays. We can specify priorities for different overlays, and then allow ODIN-S to allocate computation and bandwidth across the network to respect priorities. The key features of ODIN-S include a common middleware runtime supporting multiple overlay logics, and "filters" for throttling, ordering and dropping messages in order to manage resources. We present experimental results that demonstrate ODIN-S's ability to manage resources between different types of overlapping overlays.

1 Introduction

Middleware-level overlays have proven to be a useful abstraction for building scalable distributed systems. Many different kinds of overlays have been designed and built; a small sample includes [12, 14, 22, 28, 27, 29, 33, 36]. Each of these overlays has strengths and weaknesses, and each aims for different design goals, which means that different kinds of overlays are useful for different applications. As a result it is unlikely that all applications will be built on one, general purpose overlay. Instead, there are likely to be many overlays using the same infrastructure resources, each deployed for a different purpose. For example, an enterprise might deploy a Narada overlay for message brokering, a super-peer overlay for information discovery, a Chord overlay for LDAP directory services, and so on. A simple example of two such overlapping overlays is shown in Figure 1. Several recently developed overlay toolkits have the ability to deploy overlapping overlays, including P2 [23] and GridKit [18].

It is important to mediate resource usage between all these overlapping overlays. A particularly resource intensive overlay should not starve other overlays that are not as greedy. Moreover, in many cases we want to assign priorities to overlays, and to give more resources to higher priority overlays (but again, without starving lower priority overlays.) For example, an enterprise may give the highest priority to the messaging overlay that is supporting its day-to-day operations, and less priority to an information discovery overlay that merely supplements its internal document search apparatus.

How can we mediate resource usage between overlapping overlays? We must allocate both bandwidth resources and processing resources in a fair but prioritized way between overlays. Existing overlay toolkits lack the ability to trade off resources between multiple overlays, or do not enforce fine grained priorities over all of these resources. In

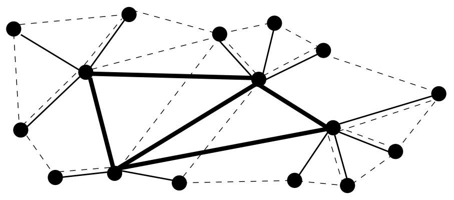


Fig. 1. Overlapping overlays. Solid lines represent a super-peer overlay; thick lines for connections between super-peers, and thin lines for connections to leaf-peers. Dashed lines represent a flat mesh network, such as a Narada broker network or unstructured peer-to-peer overlay.

this paper, we present a middleware system we have built, called *Overlay Dynamic Information Networks - Shared* (ODIN-S), that manages this resource mediation. The key architectural aspects of ODIN-S are (1) a common middleware runtime that supports the logic for multiple overlay clients on a single host, and (2) filters that manage the sending and processing of messages by these clients to enforce fairness and resource quotas.

Filters are the primary mechanism in ODIN-S for trading off resources between overlays. Filters can be used to throttle, schedule and drop messages to enforce quotas and overlay priorities. As such, filters must be used both to filter incoming messages and outgoing messages sent by a peer. We describe how to construct filters to manage the processing load, upload bandwidth and download bandwidth for a peer.

The primary contribution of this work is an architecture that integrates and adapts multiple techniques from other domains for the purpose of mediating resource usage among overlapping overlays. For example, we apply a weighted fair queuing discipline [17, 16], typically used in network routers, to the problem of scheduling messages that are delivered and sent by the middleware. We develop an adaptive algorithm, inspired by a similar algorithm used in database replication [26], to allocate download bandwidth among multiple upstream peers. We use the concept of ingress and egress filters, typically used at network boundaries (for example, to detect and defeat denial of service attacks) to manage message flows between individual peers. In this paper, we describe how to combine and extend all of these techniques into a comprehensive middleware system for managing the resource usage of multiple overlays.

The remainder of this paper is organized as follows. In Section 2, we place ODIN-S in context with related work. Section 3 describes the overall architecture of ODIN-S, focusing on the support for overlapping overlays. In Section 4 we demonstrate how to use filters for a variety of resource management goals. Section 5 presents experimental results, and we conclude in Section 6.

2 Related work

General scheduling of resources for data flows is a well known problem. Queuing disciplines for scheduling packets have been studied in depth in the networking domain [16, 17, 6]. Scheduling of flowing data has been studied both in operating systems research [9] and database research [3]. We view the traffic on an overlay as a data flow, and then adopt and extend techniques from different domains to manage the resources used by this traffic. Queuing of traffic from a single overlay is used in data stream systems such as Borealis [13] and overlay toolkits such as P2 [23]. We generalize this approach to queue messages from multiple overlays to enforce priorities across general overlay types. Some recent work has been done on scheduling for overlays, such as operator scheduling for distributed data stream processing [32, 27, 13], or load-based rearrangement of query streams in peer-to-peer overlays [25]. Our work generalizes these approaches for arbitrary, and overlapping, overlays.

In addition to throttling and scheduling, ODIN-S's filters can be used for load shedding, dropping messages when buffer resources are saturated. This approach is used to drop packets at overloaded routers [19], and tuples in overloaded data stream processors [15]. Our current implementation provides only simple tail drop of messages, but other policies could be easily added just by implementing a new filter. Moreover, it may be desirable to implement application-level endpoint congestion control (such as the congestion control schemes used in TCP) in addition to the in-network filters; we have not yet explored this approach.

Our upstream/downstream filter sets distribute a fair queuing algorithm among multiple upstream filters. A similar approach is taken in Core Stateless Fair Queuing (CSFQ) [34], where packets entering a core network are labeled by multiple upstream edge nodes to achieve approximate fair queuing behavior inside the core. Unlike the "core-stateless" approach where the core has no fair queues, ODIN-S can place filters at all overlay peers, ensuring per-link fair queuing and finer-grained control over resource usage. Other approaches to network QoS include Integrated Services (IntServ), in which flows must reserve resources that will be needed [10], and Differentiated Services (DiffServ), in which classes of flows (say, from the same ISP) are provided service based on a service level agreement [8]. Our techniques borrow the notion of using control messages to configure resource management from these approaches. Furthermore, we integrate aspects of each approach: our per-peer filters are similar to the per-router states maintained in IntServ, while our approach to managing the traffic of an entire overlay (as opposed to the flow of a single request) is similar to the classes of flows in DiffServ.

Packet and message filters are used in a variety of systems for different purposes. Examples include security filters at network boundaries, message filters in Web middleware such as mod_perl, content filters in publish/subscribe middleware [5], and so on. We apply the concept of filters to provide pluggable middleware components for managing resources in overlapping overlays.

Orthogonal to our approach to overlapping overlays is to have distinct overlays interoperate, as in PPPP [4]. In this approach, the overlays are using different hardware resources, and thus resource mediation is not as important.

3 Architectural support for overlapping overlays

In this section, we describe the architecture of ODIN-S, focusing particularly on the features that support overlapping overlays. ODIN-S is a middleware toolkit designed to support various types of overlays, including unstructured peer-to-peer overlays [25,

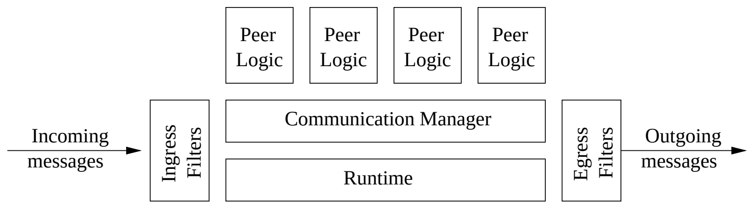


Fig. 2. ODIN-S peer architecture.

36], structured overlays [33, 28, 31], message and event dissemination overlays [14], data stream transport and processing overlays [27], and so on. In each of these overlay types, many distributed "peers" or "nodes" are connected by middleware-level, logically persistent communication "links." For clarity in the rest of our discussion, we will refer to "peers" to mean either "peers" or "nodes."

The architecture of an ODIN-S peer is shown in Figure 2. We now describe each of the components of the architecture. As a running example, we will use both a super-peer network and a completely unstructured overlay (e.g. as in the original Gnutella protocol). There are many other types of overlays, but these relatively simple overlays are used to clarify the discussion. In a super-peer network, "leaf-peers" send a summary of their content to "super-peers." Super-peers, which typically are high capacity nodes, handle all of the searching, both looking for matches in the summaries of their leaf peers and forwarding messages to other super-peers. Leaf-peers are thus unloaded. An unstructured network has only one type of peer, and each peer connects to some number of neighbors. Each peer processes searches over its local content, and forwards the search to some or all of its neighbors. The peer can "flood" the search (as in the original Gnutella) by sending it to all of its neighbors, or may choose a more efficient routing strategy [1, 24, 35, 21, 11].

3.1 Peer logic

A *peer logic* handles the routing and topology management for a single overlay. A peer that is participating in multiple overlays will have multiple peer logics, one per overlay. In order to join a new overlay, the peer creates and starts a new peer logic. Similarly, to leave an overlay, the peer stops and destroys a peer logic.

A peer logic in ODIN-S is comprised of two sub-components: the *routing logic* and the *topology manager*. The routing logic receives messages from other peers, processes them, and decides which neighbors (if any) to forward the messages. For example, a routing logic for a super-peer will receive search messages from leaf-peers and other super-peers, look for matching content in its indexes, return result messages if any content is found, and forward the search messages to other super-peers. In contrast, a topology manager manages the set of neighbors that the peer has in the overlay, by making and breaking connections to other neighbors based on the overlay's neighbor policy. For example, a topology manager for a super-peer will always try to have at least N super-peer neighbors, making connections to new neighbors if existing neighbors leave the overlay. Similarly, the super-peer topology manager will accept connections from leaf-peers, possibly enforcing some upper limit on the number of connected leaf-peers. By separating the peer logic into routing and topology components, we can easily mix and match different routing and topology algorithms in different overlays. These reusable components makes it easier to extend ODIN-S to support different overlays.

Peer logics in ODIN-S also provide some key functionality available in other overlay toolkits [23, 30]. For example, peer logics can set and respond to timers, and pass events to and from the application (such as a GUI or other application logic.)

3.2 Communication manager

The communication manager handles the setting up/tearing down of connections, and the sending/receiving of messages, for all peer logics residing at a peer. The communication manager API allows peer logics to create connections to other peers and send messages, abstracting away the details of the underlying transport layer. Thus, the same unmodified peer logic could be used over a variety of underlying transports, including raw TCP sockets, SOAP calls, JXTA connections, and so on. Our system currently uses TCP sockets.

The communication layer also provides multiplexing and demultiplexing of messages from different peer logics over the same underlying transport. Each overlay is identified by an integer *overlay ID* that is a constant value across the entire overlay. This overlay ID is included in the header of each message sent, so that the receiving communication manager can dispatch it to the appropriate peer logic.

In this way, the overlay ID "names" the overlay. For example, we could have different but overlapping super-peer networks if each super-peer network was identified by a different overlay ID. Creating a new overlay involves choosing a new overlay ID, starting a peer logic for that overlay ID at some peer, and publicizing the overlay ID so that other peers can join (for example, by advertising it on a registry). The only requirement is that logically different overlays have different IDs (no *overlay ID collisions*). If the overlays exists within a single organization, the organization's IT department can hand out overlay IDs. On the Internet, the problem of avoiding ID collisions is somewhat more complex. There might be an ICANN-style service for handing out overlay IDs, or IDs may be cryptographic hashes of some meaningful description string. If we choose a large enough ID space (e.g., 256 bits), then we can even choose overlay IDs randomly and have minimal chance of collisions.

3.3 Runtime

The runtime provides the base functionality for each peer. The runtime creates and destroys peer logics, schedules timers, dispatches events between the application layer and the peer logics, provides a logging facility, and provides other services. Creating a peer means starting the runtime. The runtime will then construct a communication manager. The runtime will also accept "create peer logic" events, and create the appropriate peer logic according to a set of properties embedded in the event. The runtime is thus like a simplified application server that creates and destroys individual components as needed.

3.4 Filters

The resource mediation capability in ODIN-S is provided by *filters*. In particular, filters manage messages according to resource quotas and policies. Filters provide:

– *Throttling* of message rates to enforce quotas,
– *Ordering* of message sending and delivery to enforce priorities, and
– *Dropping* of messages to shed load when necessary.

For example, a filter might enqueue (and hence delay) messages to perform throttling, prioritize the queue to reorder messages, and drop new messages if the queue of existing messages reaches a pre-defined limit.

Two types of filters can be created. An *ingress filter* filters incoming messages before they are delivered to the appropriate peer logic. An *egress filter* filters outgoing messages that are generated by a peer logic, before they are actually handed over to the transport layer for sending. Both ingress and egress filters are installed in the communication manager and shared across peer logics, so that one filter can manage resources shared across overlays. For example, an ingress filter can enforce a quota per minute on the overall number of messages processed by the peer for any overlay; any incoming messages over the quota in a given minute would be enqueued, regardless of which peer logic they were destined for. At the same time, filters can be used to enforce policies for a particular overlay, passing through (without filtering) any message on other overlays.

Filters are pluggable components, so that multiple ingress and egress filters can be installed at a peer. For example, one egress filter might enforce an overall quota on the upload bandwidth used, while another egress filter might be used to throttle the message rate of a particular overlay. Filters process messages in the order they were installed, and only if all filters approve a message will the message be delivered/sent (respectively, for ingress/egress filters). If a filter delays a message, when it is eventually approved by that filter it will be passed to the next filter in the list before delivery/sending. Filters can also be created dynamically, as needed. For example, when a peer in one overlay joins a second overlay, it might dynamically create a filter to manage the two overlays.

Filters do not have to be passive components, responding only to incoming or outgoing messages. Filters can also set timers, and be notified when the timer expires. In addition, filters send messages to communicate with filters on other peers. In this case, the filter receiving the message should install itself as an ingress filter so it can receive the message and avoid its dispatch to any peer logic. (Because the same filter object instance can be installed as an ingress and egress filter, this mechanism allows even egress filters to communicate with each other.)

The generality of our filtering architecture means that filters can also be used to perform other functionality for overlays, such as gathering statistics about traffic, filtering out ill-formatted messages, or logging messages for recovery purposes. Such functionality is outside of the scope of this paper, and we focus on using filters for resource mediation here. We examine specific examples of using filters to mediate resources among overlapping overlays in the next section.

4 Mediating resources using filters

We now illustrate how filters in ODIN-S can be used to mediate different types of resources shared among overlays. We focus on techniques for mediating *processing load*, *upload bandwidth* and *download bandwidth*. A typical peer will want to manage its overall load, and thus will likely install filters simultaneously for all three goals. The techniques presented here are representative examples; filters are a general architectural feature that can be used to implement a wide range of load and resource management techniques. Experiments in Section 5 demonstrate that the filters described here are effective at trading off resources between overlapping overlays.

4.1 Processing load - priority-based ingress filter

A machine hosting multiple overlays can experience CPU overload as it processes all of the messages arriving on all of the overlays. Therefore, it is important to be able to limit the amount of CPU used by the ODIN-S middleware. This limit can be enforced by operating system mechanisms, such as UNIX “nice.” Another approach is to create an ingress filter that enforces a limit on the messages processed per unit time. One advantage of the ingress filter-based approach is that the limit can be expressed in high-level terms (e.g., “process no more than 10 search messages per second”). A disadvantage of enforcing an absolute quota using a filter is that if the machine is otherwise idle, the spare CPU cycles will not be used by ODIN-S. Most likely, combining OS priority and filter-based quotas will be useful.

In either case, ODIN-S will be given a limited amount of processing capacity in which to handle messages. This capacity must be allocated to different peer logics in accordance with the priority of their overlay. For example, if a machine is participating in a high priority super-peer overlay and a low priority unstructured overlay, then proportionally more processing capacity should be given to the messages from the super-peer overlay (without starving the unstructured overlay.) We assume that each overlay is assigned a global priority, for example by the enterprise using the overlays. If different peers have different notions of priority for different overlays, then they can use ODIN-S filters to enforce those priorities locally. However, in this case, there will be no global enforcement of priority, which is appropriate given that there is no global agreement on overlay priority.

A filter that implements queuing can enforce both an absolute quota on messages processed (e.g., throttling) as well as enforcing priority (e.g., message ordering.) Arriving messages are automatically enqueued. Messages are dequeued and delivered to the appropriate peer logic in an order determined by priority. If the filter is enforcing a quota, then messages are only delivered periodically. For example, if the quota is 10 messages per second, then a new message will be taken off the queue and delivered every 100 milliseconds. If the filter is only enforcing priority (and not a quota), then a new message is taken off the queue as soon as the previous message has been processed. The filter can also enforce a maximum queue size, and drop messages according to some drop policy when the queue is full.

Priority-based ordering of messages An important consideration is the order in which we dequeue messages. A simple approach is to always dequeue the message from the highest priority overlay. However, as is well known from experience with network routing and operating systems scheduling, such an approach can lead to starvation for lower priority processes. Another possible approach is to divide the quota among overlays according to priority, for example giving 1/3 to the low priority overlay and 2/3 to the high priority overlay. However, this approach is not *work-conserving*: when the super-peer network is not using all of its quota, the unused portion is wasted, when it should be given the unstructured overlay. Instead, we need a fair and work-conserving algorithm for dequeuing messages.

These properties are provided by a class of *weighted fair queuing* (WFQ) algorithms, traditionally used to schedule packets in network routing. WFQ allocates an overloaded network channel to a flow in proportion to the flow's relative priority. Thus, if the sum of flow priorities is P, and a particular flow has priority p_i, that flow should receive p_i/P of the channel bandwidth. WFQ is work conserving because it schedules packets eagerly: if there are no queued packets for some other flow j, then WFQ schedules more of flow i's packets, beyond i's guaranteed bandwidth proportion of p_i/P. We adapt the WFQ approach to CPU scheduling in ODIN-S: if an overlay i has priority p_i, and the sum of priorities over all the overlays the peer participates in is P, then the overlay i should receive p_i/P of the CPU (or of the CPU quota, if one is enforced.) Practically, this means that if we dequeue P total messages in a time period, p_i of those should be from overlay i. If some overlay is not using its full allocation, the unused portion is fairly divided among the remaining overlays, again respecting priority.

WFQ was proposed originally in [16]. We actually use a follow-on proposal called *start-time fair queuing* (SFQ) [17] in ODIN-S. SFQ has two key advantages over the original WFQ algorithm: 1. SFQ has less computational complexity than WFQ, and 2. SFQ is more fair when the sender's rate is not constant. SFQ also has advantages over other queuing disciplines; these advantages are outlined by Goyal, Vin and Cheng [17]. Other queuing disciplines can be enforced by implementing an appropriate filter.

Briefly, SFQ operates as follows. (For more details, see [17]). Each message m that is enqueued is given a start tag S_m and a finish tag F_m, and messages are dequeued and serviced in order of increasing S_m (with ties broken randomly). The start tag S_m is set equal to either the start tag of the message that was being sent when m arrived, or the finish tag of the previous message enqueued for m's overlay, whichever is greater. The finish tag F_m is set equal to $S_m + c_m/p_m$, where c_m is the cost of sending the message, and p_m is the priority of m's overlay. The c_m value can be the length of the message in bytes, the estimated processing time, or some constant (if all messages are roughly equally expensive to process.) In this way, successive messages on the same overlay are given progressively higher finish (and hence start) tags, while messages on different overlays are given finish (and hence start) tags that are interleaved proportionally to their c_m/p_m ratio, resulting in proportionally more service for higher priority messages.

4.2 Upload bandwidth - priority-based egress filter

Another resource used by the peer is upload bandwidth (link capacity used for sending messages). With asymmetric connections (such as residential DSL), the upload band-

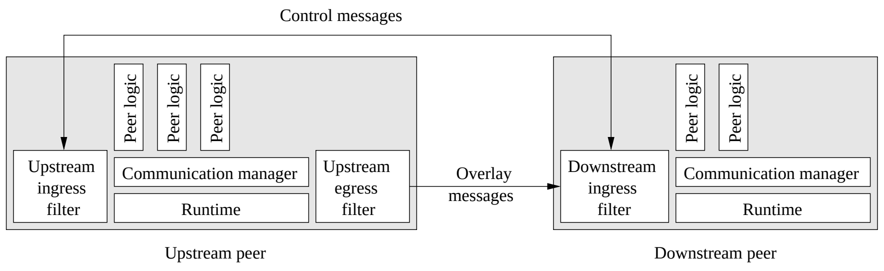


Fig. 3. Upstream/downstream filter set.

width may be different than the download bandwidth. Thus, we often have to manage the upload bandwidth separately from download bandwidth. Even with symmetric connections (e.g. a machine connected to a LAN via Ethernet), it is important to manage the number of messages sent in order to mediate the usage of the link bandwidth.

Upload bandwidth is managed by an egress filter. When any peer logic attempts to send a message, the message is automatically enqueued. The filter takes messages off the queue for actual sending according to the overlay priority. The filter can also enforce a quota on the upload bandwidth used by sending (dequeuing) no more than N bytes in any time unit. In fact, the same priority based filter described in Section 4.1 can be used, except that the filter is installed as an egress filter instead of an ingress filter. Therefore, we can use SFQ (or any desired queuing discipline) here as well.

4.3 Download bandwidth - upstream/downstream filter set

Download bandwidth is more difficult to manage than upload bandwidth, which can be managed by installing a single egress filter. In contrast, a peer cannot manage its download bandwidth by installing a single ingress filter; by the time the ingress filter touches the arriving message, the download bandwidth has already been used. Instead, a peer needs to cooperate with its neighbors in order to manage its download bandwidth. Consider a peer i that wishes to cap the amount of download bandwidth used for overlay messages. Peer i can contact each of its neighbors, regardless of which overlays they participate in, and ask them to throttle their message sending rate so that the total bandwidth used is no more than the cap.

Figure 3 shows an *upstream/downstream filter set* that implements the cooperative management of download bandwidth. The upstream filter is installed at each of peer i's neighbors, while the downstream filter is installed at peer i itself. The downstream filter determines how much bandwidth can be used by each upstream neighbor, and sends control messages to the upstream filters to ask them not to use more bandwidth. Both the upstream and downstream filters are ingress filters so that they can receive control messages. However, the upstream ingress filter creates egress filters on the upstream peer to enforce the downstream peer's bandwidth limitation requests. For example, if peer i asks peer j to send no more than 10 Kbps, then peer j will install an egress filter with a quota of 10 Kbps. This egress filter will apply only to peer i, so messages sent by peer j to peers other than i will be passed through the filter without delay. In fact, peer j may create multiple egress filters, one per downstream neighbor that requests a cap. However, peer j would only have a single instance of an upstream ingress filter to respond to control messages and manage all of the egress filters.

Note that since connections in an overlay can be symmetric, j may be downstream of i as well as upstream. In this case, j will also have a downstream filter and i will have an upstream filter, and i will install egress filters at j's request (just as j installs them at i's request.) Then, the total complement of filters at both i and j would include: a upstream ingress filter, a downstream ingress filter, and quota-enforcing egress filters.

If two peers are neighbors in multiple overlays, then the upstream egress filter must enforce overlay priority as well as a download quota when determining which messages to send to the downstream peer. To do this, the egress filter created by the upstream filter can be the same filter as described in Section 4.2: the priority-based egress filter, perhaps based on SFQ.

Allocating bandwidth quotas to upstream neighbors Consider a peer i that wants to enforce a total quota of D bytes per second on its download bandwidth. Peer i must ask its upstream peers to throttle their message sending so that the total bandwidth is no more than D for all messages sent by i's neighbors. The simplest way to do this is for i to divide the quota equally among all of its neighbors: if i has n neighbors, then each neighbor is given a quota of D/n. However, this simple approach may waste quota, since the same quota is given to each neighbor regardless of the number of overlays the neighbor wants to send traffic on. Neighbors that want to send traffic on one or only a few overlays will have a larger quota than is necessary. Similarly, it makes no allowance for the fact that some neighbors may only be members of low priority overlays, and thus should not get an equal allocation of the quota as neighbors who are members of higher priority overlays.

We now describe how to allocate the quota D to upstream neighbors according to the priority of overlays those neighbors are participating in. Our algorithm divides the total quota proportionally based on the priority of overlays, and then further divides these fractional quotas evenly among the neighbors that wish to send traffic on a particular overlay. The algorithm operates as follows. Peer y tracks which upstream neighbors want to send traffic on each overlay, either by tracking received messages or by having upstream peers send control messages listing the overlays on which they want to send messages. Periodically, peer y adjusts the quota assigned to each upstream neighbor as follows. First, it divides the total quota into per-overlay fractional quotas, proportional to the relative priority of each overlay: each overlay i is given quota $d_i = D \times \frac{p_i}{P}$. These fractional quotas are then evenly divided among the neighbors that wish to send traffic: if there are c_i neighbors sending traffic on overlay i, each neighbor receives a slice of the fractional quota equal to $\frac{D}{c_i} \times \frac{p_i}{P}$. The total quota given to an upstream peer is the sum of the slices for that peer. That is, define $t_{i,j}$ as 1 if neighbor x_j wants to send traffic on overlay i, and 0 otherwise, and m as the number of overlays. The quota assigned to an upstream neighbor x_j is $\sum_{i=1}^{m} t_{i,j} \times \frac{D}{c_i} \times \frac{p_i}{P}$. Each upstream neighbor manages its quota using an egress (SFQ) filter.

As an example, consider a peer Y, with a download quota of 9 messages/second, and its three upstream neighbors A, B and C. A and B both want to send traffic on overlay 1. C wants to send traffic on both overlay 1 and overlay 2. Assume that overlay 2 has twice the priority of overlay 1. Then, the total quota on messages delivered to Y from any peer should be divided into one-third for overlay 1 and two-thirds for overlay 2. The quota for overlay 1 (3 messages/second) will be divided evenly among A, B and C (each receiving 1 message/second). The entire quota for overlay 2 (6 messages/second) will be given to C. The total quota given to C will be 7 messages/second.

It is also possible to extend this algorithm to give more quota to upstream neighbors that wish to send more traffic on a given overlay. Then, instead of dividing the fractional quotas into equally sized slices, we would divide the fractional quota based on the observed traffic rate from each neighbor. However, experiments in Section 5 demonstrate that in practice, this approach does not work as well as the equal division in the algorithm detailed above.

5 Experimental results

We have conducted experiments to determine how effective the ODIN-S architecture is at fairly managing resources between overlapping overlays. Our experiments utilize ODIN-S components both in a discrete event simulation, and as actual peers. Our simulator allowed us to quickly examine many overlay combinations and parameter settings, while the prototype allowed us to examine the real system in action. It is important to note that the peer logic and filter implementations were the same in the simulator and prototype; in other words, in the simulations each peer was processing and sending actual messages. This philosophy of simultaneously implementing a simulator and peer client software using the same components was proposed in [30, 20]. For the experiments in this paper, we used the peer implementation to calibrate and validate our simulations. Here, we report simulation results.

The primary metric we have considered is *throughput*, measured in terms of messages processed by the overlay per second. Although we are examining different overlays providing different functionality, this throughput metric allows us to examine a common performance measure across all overlays.

5.1 Experimental setup

We experimented with three types of overlays: an *unstructured peer-to-peer overlay*, with searches routed via optimized random walks [1, 24], a *super-peer overlay*, with searches flooded between super-peers [36], and a *multicast tree overlay*, with a single source multicasting updates to multiple clients. The multicast overlay is modeled on the end-system multicast trees provided by the Narada system [14], although we have only implemented the multicast tree construction and not yet the topology-aware optimizations that Narada provides.

In most of our experiments, there were 1,000 total peers participating in multiple overlays. Each peer remained alive during the entire experiment. This scale and lack of peer “churn” represents enterprise-scale overlays. A typical Internet peer-to-peer overlay would likely have more peers and more churn. Given the priority-based nature of our techniques, and the need to use a common runtime, we expect enterprises or other similar organizations to be the most immediate target of our techniques. However, we also examined the scalability of our techniques to larger networks: due to space restrictions, we do not report the results, which were consistent with the results for 1,000 peers. We also examine the effectiveness of our techniques in the presence of peer churn (in Section 5.5).

For the unstructured and super-peer overlays, we downloaded HTML data from real web sites, and each peer provided full-text keyword search over the data from one website (using standard IR techniques: scored using a normalized cosine distance of TF/IDF weighted queries and documents, and a document match had score > 0.1). If a peer participated in multiple overlays, each peer logic searched documents from a different site. We used real keyword searches from the search.com search engine. In the super-peer network, only high capacity peers (capacity greater than the average) chose to become super-peers, with a probability 0.1. In the multicast overlay, a single source generated small (< 100 bytes) updates, and propagated the updates along the multicast tree. Each multicast peer had up to five children in the tree.

Each peer was given a quota, measured in messages/second, that it could receive, process, and transmit. Alternatively, the quota could have been measured in message size, time to process, etc. We chose number of messages both because it generalizes across bandwidth and CPU time (which would otherwise be measured in different units), and different types of overlays (which would have different processing costs, message sizes, and so on). To determine appropriate quotas for peers, we measured the maximum throughput achievable with our current prototype. We started an unstructured overlay using real ODIN-S peers, each on separate, otherwise unloaded machines connected by gigabit Ethernet. Each machine had dual 3.0 GHz Pentium4 Xeon CPUs and 4 GB RAM. The maximum throughput measured was 135.5 messages per second. We expect this type of machine to be in the mid to high range for overlay peers; some machines (e.g. servers) will be more powerful while many will be less powerful (both in terms of CPU and bandwidth.) Therefore, in our experiments, our simulation models machines as having capacity randomly chosen in the range 20-200 messages per second. Of course, machines that are conducting other work besides hosting an overlay node may decide to set a quota less than their maximum capacity. In such a situation, the absolute values of the quotas would be less but the range (e.g., an order of magnitude) would likely be similar.

5.2 Overlapping unstructured overlays

Our first set of experiments measured the throughput provided by ODIN-S for overlapping unstructured overlays. We started with unstructured overlays because they are the most traffic-intensive of the three overlay types we implemented, and thus they give a sense of how the system performs under heavy load. Other overlay types are considered in the next section. The experiments here examine two overlapping overlays; in other results (not shown here) we have increased the number of overlapping unstructured overlays and observed similar results.

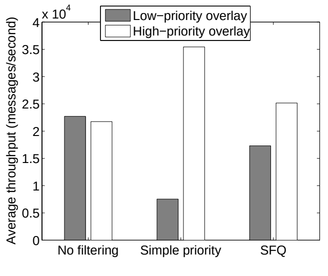


Fig. 4. Throughput without filtering, with simple priority filtering, and with SFQ filters.

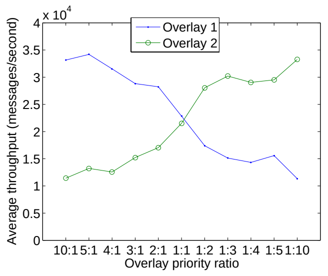


Fig. 5. Throughput with varying priority ratio.

Ingress filters for managing processing load We created two overlapping unstructured overlays: a *low priority* overlay, and a *high priority* overlay, with priority equal to twice that of the low priority overlay. Each peer participated in both overlays. We ran the system for 100 simulated seconds, generating a total of 25,000 search requests per overlay during this time. In simulations of peers with unlimited capacity, this setup results in an average of 99,000 messages per overlay per second (across the whole overlay). However, the limited capacity of peers sharply reduces the actual throughput.

Figure 4 shows the average throughput for no ingress filtering, ingress filtering using a simple priority scheme (always prefer higher priority messages), and ingress filtering using SFQ. As the figure shows, in each case the total throughput over both overlays is about 44,000 messages per second. However, without filters, that throughput is allocated unfairly: the low and high priority overlays receive the same service. The throughput is also unfairly balanced with simple priority filters, as the low priority overlay is starved (7,500 messages/second) compared to the high priority overlay (35,000 messages/second). In contrast, the SFQ ingress filters result in service that more properly reflects the overlay priorities: the high priority overlay receives 60 percent of the total throughput. This result is not exactly the 2:1 ratio of high/low priority, and reflects overlay-wide queuing effects that disproportionately affect the high priority overlay. In particular, since more high priority messages are approved by filters at some nodes, there are disproportionately long queues of high priority messages at other nodes, and these queuing delays reduce throughput of the high priority overlay somewhat. Despite this issue, the ingress filter using SFQ queuing most effectively preserves the overlay priority. We also ran experiments with larger and smaller rates of search requests, and observed similar effects.

Varying priorities In the next experiment, we used the same overlays and traffic as in the previous experiments, but varied the priority ratio between the two overlays from 1:10 to 10:1 (each different ratio represents a different experimental run). As Figure 5 shows, the changing relative priority results in changing relative throughput. In the middle of the figure (for ratios near 1:1), the ratio of throughput nearly exactly reflects the ratio of priorities. For larger ratios (on the left and right of the figure), the throughput of

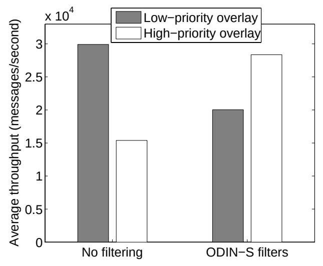


Fig. 6. Throughput when the low priority overlay has double the traffic of the high priority overlay.

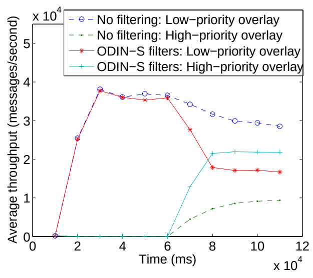


Fig. 7. Throughput when the high priority overlay starts 50 seconds after the low priority overlay.

the high priority overlay flattens, while the lower priority overlay receives less throughput. In all cases, the high priority overlay properly receives more service. The flattening results from an increasing number of "bottleneck" peers as more high priority messages are sent. Bottleneck peers are low capacity peers that have long queues and effectively throttle the throughput of the entire overlay. Thus, even as more priority is allocated to the overlay, the network simply cannot provide it more service. (This effect is the reason the SFQ filter provides only a 60/40 throughput allocation in the previous section.)

Varying traffic rates So far, both the high and low priority overlays have had the same, constant rate of traffic to send. We also experimented with cases where the traffic rates varied between overlays and over time.

First, we ran an experiment where the low priority overlay had twice the query rate of the high priority overlay. Figure 6 shows the results. Without filtering, the low priority overlay receives significantly more service than the high priority overlay. ODIN-S filters properly preserve priority; even though the low priority overlay has more traffic to send, it still receives less service than the high priority overlay.

Next, we ran an experiment where both overlays had the same query rate, except that the low priority overlay experienced a load spike, doubling its traffic for 20 seconds (starting at 50 seconds.) The results (not shown) indicate that the high priority overlay is not affected, and continues to receive higher service than the low priority overlay.

Third, we ran an experiment where both overlays had the same query rate, but the high priority overlay was created halfway through the simulation. This models a scenario where an overlay exists, and then another overlay is started using the same machines. As Figure 7 shows, initially the low priority overlay is receiving high service, because it is the only overlay. In the no-filtering case, the low priority overlay continues to receive the most throughput even after the high priority overlay begins to transmit traffic. In contrast, with ODIN-S filters, when the high priority overlay starts, it quickly achieves higher throughput than the low priority overlay.

These results all show that ODIN-S filters can effectively trade off resources between overlays, even when the traffic rates change.

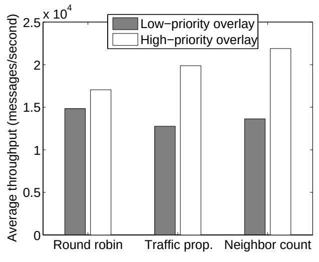


Fig. 8. Approaches to allocating quota to upstream peers.

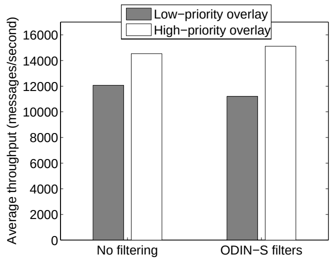


Fig. 9. Two overlapping multicast overlays.

5.3 Upstream/downstream filter sets for managing link bandwidth

Next, we examined the effect of our adaptive algorithm for allocating quota to upstream egress filters. We used the same overlapping unstructured overlays and message load as in the previous section. We compared three methods of allocating upstream quota:

– *simple round-robin*: the quota was divided evenly among upstream neighbors.
– *adaptive, neighbor count*: our algorithm from Section 4.3 was used to adjust upstream quota based on which overlays at neighbors participate in.
– *adaptive, traffic-proportional*: we extended our adaptive algorithm to adjust upstream quota based on the traffic rates at upstream neighbors.

The traffic-proportional algorithm gives more quota to neighbors that wish to send more traffic. In this case, the upstream neighbor measures its traffic rate and sends updates to the downstream neighbor. The total quota is divided proportionally based on overlay priority into fractional quotas, and then further divided proportionally to the amount of traffic that the upstream neighbors of each overlay want to send. In all cases, the dynamically created upstream egress filters were SFQ filters.

Figure 8 shows the results. As the figure shows, in the simple round-robin method, the throughput experienced by both overlays is roughly equal. In contrast, both adaptive algorithms allocate total throughput according to priority, such that the high priority overlay receives more throughput than the low priority overlay (again, split 60/40). Another key difference between the three techniques is that the adaptive algorithms provide greater total bandwidth over both overlays than the simple round robin. In fact, the adaptive, neighbor count algorithm results in the highest total throughput, with 9 percent more total throughput than the traffic-proportional algorithm, and 11 percent more total throughput than the round-robin approach. Round-robin wastes quota by allocating the same amount to neighbors that host only one overlay as to those that host multiple overlays. The traffic-proportional approach would seem to effectively allocate quota based on traffic rates (and in fact was our first approach.) However, our experiments show that this approach also wastes quota when traffic is bursty. A burst causes a large quota window to open on one neighbor, consequently reducing the quota on other neighbors. When the burst is over, another burst typically arrives at another neighbor, who now has a small quota. The result is that the large quota on the formerly bursty neighbor is wasted until the adaptive algorithm reallocates quota. The neighbor count approach provides more stable quotas, but does so in recognition of the actual overlays of neighbors, resulting in both fairness and larger total throughput.

5.4 Other overlay types

In addition to unstructured overlays, we also examined super-peer and multicast overlays. First, we present results where the overlapping overlays are of the same type, and then we present results where the overlays are different.

Overlapping overlays of the same type We examined a network with two overlapping super-peer overlays. In this case, super-peers handle almost all of the overlay messages, since search messages are never sent to leaf peers. Therefore, there is only contention for resources when the same peer hosts super-peers for both overlays. We again generated 25,000 searches per overlay. The results (not shown) demonstrate that at the nodes where there is resource contention, the higher priority overlay receives higher throughput, consistent with the relative priorities between the two overlays.

Next, we ran an experiment with two overlapping multicast trees. Each multicast source generated 25,000 events. The high priority tree had twice the priority of the low priority tree. Figure 9 shows the resulting throughput. As the figure shows, the effect of filtering is less prominent. Although filtering results in the high priority overlay receiving more throughput, the allocation is not as fair: the high priority overlay only receives 57 percent of the throughput. In an application-level multicast tree, the total throughput is heavily dependent on the capacity of the root peer and peers near the root, since they are the bottlenecks for dissemination to the rest of the tree. If a peer is a bottleneck node in both multicast overlays, then ODIN-S filters can mediate the resources at the bottleneck; otherwise, the filters have only a minor effect. The result is that ODIN-S filters allocates total throughput more effectively than in the no filtering case, but not as well as for other types of overlays.

Because performance of the multicast overlay depends so heavily on the topology and nature of the bottleneck nodes, to ensure an apples-to-apples comparison we had to hardwire our system to construct the same overlay topology in both the filtering and no filtering scenario. We do the same for multicast tree experiments in the next section.

Overlapping overlays of different types We examined four combinations of heterogeneous overlapping overlays: (1) unstructured + super-peer, (2) flooding-based unstructured + random-walk based unstructured, (3) super-peer + multicast, and (4) unstructured + multicast. These experiments are just a subset of all of the possible combinations of overlays. However, they allow us to examine how well ODIN-S trades off resources between overlapping overlays of different types.

First, we ran an experiment where an unstructured overlay overlapped with a super-peer overlay. We assigned the super-peer overlay twice the priority of the unstructured overlay. In this case, the super-peer overlay's resource requirements are concentrated at super-peers (approximately 5 percent of the total nodes), which have high traffic volumes. As shown in Figure 10, without priority filtering, the super-peers become a

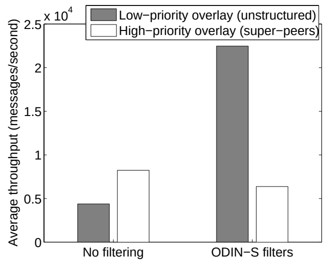


Fig. 10. An unstructured overlay overlapping with a super-peer overlay.

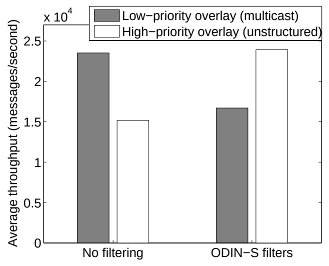


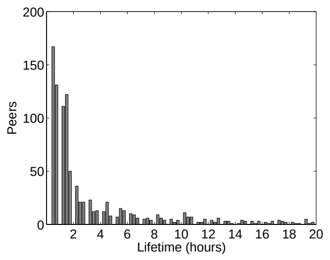
Fig. 11. An unstructured overlay overlapping with a multicast overlay.

bottleneck for both overlays, effectively starving the unstructured overlay. In contrast, with ODIN-S filtering, the unstructured overlay is given its fair share of priority at the super-peers, reducing the bottleneck effect and resulting in more than a factor of five increased throughput. This demonstrates how ODIN-S can prevent the topology features and hotspots of one overlay from impeding the throughput of the other overlay. Note that the super-peer overlay, which is high priority, sends relatively few messages, since it is more efficient than the unstructured overlay. However, at peers hosting a super-peer, the super-peer overlay is receiving twice the service of the unstructured overlay, without starving the unstructured overlay.

Next, we examined two different types of unstructured overlays: a flooding-based unstructured overlay (as in the original Gnutella) and a random-walk based overlay. Flooding, which is not scalable for Internet overlays, might be used in enterprise-scale overlays because it offers low result latency. For this reason, some domain-specific tools use a flooding-based overlay [2]. Our results (not shown) indicate that despite the difference in routing method, ODIN-S filters result in both overlays receiving throughput in proportion to their priority. Without filtering, both overlays receive approximately the same throughput, despite their priority difference.

We also examined a super-peer overlay overlapping with a multicast overlay. The results (not shown) are similar to the case of the two overlapping multicast trees (Section 5.4): filtering improves the fairness of the allocation of throughput, but the effect is not as dramatic as in the other overlay types. Again, the unique topology features of the multicast tree (bottleneck nodes are near the root) means that unless a bottleneck node and super-peer are hosted on the same peer, ODIN-S filters have limited effect.

Finally, we examined an unstructured high priority overlay overlapping with a low priority multicast tree. The results are shown in Figure 11. As the figure shows, ODIN-S has a large effect on the throughput, effectively allocating more throughput to the high priority overlay, while no filtering allows the low priority multicast tree to grab most of the service. In this case, the bottleneck nodes of the multicast tree are always overlapping with a traffic-bearing node in the unstructured network (since all peers in unstructured network forward traffic.) Then, ODIN-S can effectively allocate the throughput at the bottleneck nodes between the two overlays, resulting in an overall fair allocation of throughput.



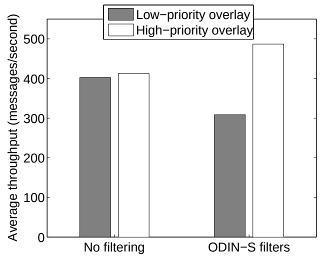


Fig. 12. Distribution of peer lifetimes in the dynamic scenario. 51 peers with lifetimes >20 hours are not shown. The maximum lifetime is 120 hours.

Fig. 13. High churn scenario: two overlapping unstructured overlays.

5.5 Dynamic scenario

We examined a scenario in which overlay peers joined and left frequently, as frequently occurs in many Internet peer-to-peer overlays. Although we primarily envision our system being used for more stable networks where priority can be effectively assigned to overlays (such as in enterprise networks), we wanted to see how a large amount of churn affected the throughput. We used a real trace of peer joins and leaves from the Overnet system [7], and extracted the first 1,000 peers from the trace (representing roughly the first 14 hours of the trace.) This trace is highly dynamic: the average peer is alive for only 5 hours, while the shortest lived peers are alive for 20 minutes. Figure 12 shows a histogram for the distribution of peer lifetimes. Each peer participated in two overlapping unstructured overlays. Since Internet peers are not usually dedicated but instead used for multiple tasks besides the overlay, peers allocated 10 percent of their capacity to the overlay; hence the capacity range was 2-20 messages/second.

Figure 13 shows the results. As the figure shows, even in the presence of high churn the system performs as before: without filtering, both overlays receive the same service, but with ODIN-S filters, the higher priority overlay receives an appropriately higher level of service.

5.6 Summary of results

We can draw the following conclusions from our results:

– ODIN-S is effective at enforcing fair allocation of resources between overlapping overlays, respecting priority but avoiding starvation.
– Using the SFQ filter as an ingress filter is effective across different priority ratios, traffic patterns, and overlay types. It is least effective for overlapping multicast overlays, where the topology characteristics often outweigh what the filtering can achieve. Even in this case, however, filtering is more fair than no filtering.
– The adaptive, neighbor-count algorithm for allocating quota to an upstream neighbor is more fair, and results in better overall throughput, than a simple round robin scheme (allocate equal quota to all neighbors), or a scheme where quota is allocated proportionally to the traffic each neighbor wants to send.

– Heterogeneous overlapping overlays can have complex interactions as hotspots in one overlay impact the throughput of another. ODIN-S filters can effectively mitigate the impact one overlay has on another.
– Our architecture is effective both for overlays with little or no churn, as well as for overlays with a high amount of churn.

6 Conclusion

We have presented ODIN-S, a middleware system for trading off resources between overlapping overlays. Our architecture can be used to mediate the resource demands of different overlays deployed to provide different functionality on the same hardware. The system demonstrates how to integrate and extend techniques from multiple domains into a comprehensive middleware toolkit for deploying and managing multiple overlays. ODIN-S provides a common runtime that supports multiple peer logics, one per overlay. Our system also provides a flexible and extensible filtering mechanism. Filters can be used for a variety of tasks, and we focus on their use for allocating resources to different overlays based on priority. We describe ingress, egress and upstream/downstream filters to manage CPU usage, upload and download bandwidth (respectively). Experiments demonstrate the effectiveness of ODIN-S at enforcing fair, priority-based sharing of resources among overlapping overlays.

References

1. L. Adamic, R. Lukose, A. Puniyani, and B. Huberman. Search in power-law networks. *Phys. Rev. E*, 64:46135–46143, 2001.
2. D. Agarwal and K. Berket. Supporting dynamic ad hoc collaboration capabilities. In *Proc. Conf. for Computing in High-Energy and Nuclear Physics (CHEP)*, 2003.
3. D. Aksoy, M. F. Franklin, and S. B. Zdonik. Data staging for on-demand broadcast. In *Proc. Conference on Very Large Databases (VLDB)*, 2001.
4. H. Balakrishnan, S. Shenker, and M. Walfish. Peering peer-to-peer providers. In *Proc. Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.
5. G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. IEEE International Conference on Distributed Computing Systems*, 1999.
6. Jon C.R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. In *Proc. SIGCOMM*, 1996.
7. R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proc. of the 2nd Int'l Workshop on Peer to Peer Systems (IPTPS)*, 2003.
8. S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. Rfc 2475 - an architecture for differentiated services. http://www.ietf.org/rfc/rfc2475.txt, 1998.
9. William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed schedule management in the tiger video fileserver. In *Proc. SOSP*, 1997.
10. R. Braden, D. Clark, and S. Shenker. Rfc 1633 - integrated services in the internet architecture: an overview. http://www.ietf.org/rfc/rfc1633.txt, 1994.
11. A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proc. SIGCOMM*, 2003.

12. Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P systems scalable. In *Proc. SIGCOMM*, 2003.
13. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.
14. Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proc. SIGMETRICS*, 2000.
15. A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proc. SIGMOD*, 2003.
16. A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. SIGCOMM*, 1989.
17. P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proc. SIGCOMM*, 1996.
18. P. Grace, G. Coulson, G. S. Blair, and B. Porter. Deep middleware for the divergent grid. In *Proc. Middleware*, 2005.
19. G. Iannaccone, M. May, and C. Diot. Aggregate traffic performance with active queue management and drop from tail. *Computer Communication Review*, 31(3):4–13, 2001.
20. M. B. Jones and J. Dunagan. Engineering realities of building a working peer-to-peer system. Microsoft Research Technical Report MSR-TR-2004-54, available at ftp://ftp.research.microsoft.com/pub/tr/TR-2004-54.pdf, 2004.
21. V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proc. CIKM*, 2002.
22. A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proc. SIGCOMM*, Aug. 2002.
23. B. T. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proc. SOSP*, 2005.
24. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of ACM Int'l Conf. on Supercomputing (ICS'02)*, June 2002.
25. Q. Lv, S. Ratnasamy, and S. Shenker. Can heterogeneity make gnutella scalable? In *Proc. of the 1st Int'l Workshop on Peer to Peer Systems (IPTPS)*, March 2002.
26. C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proc. SIGMOD*, 2001.
27. P. Pietzuch, J. Ledlie, J. Shneidman, M. Welsh, M. Seltzer, and M. Roussopoulos. Network-aware operator placement for stream-processing systems. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, 2006.
28. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. SIGCOMM*, Aug. 2001.
29. P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, 2003.
30. A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc. NSDI*, 2004.
31. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware*, 2001.
32. U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, 2005.
33. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM*, Aug. 2001.
34. I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. In *Proc. SIGCOMM*, 1998.
35. B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *ICDCS*, 2002.
36. B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proc. ICDE*, 2003.