# ISP Driven Informed Path Selection

Damien Saucez Benoit Donnet Olivier Bonaventure

June 3, 2011

## 1 Introduction

We are seeing the emergence of technologies severely challenging some assumptions that have driven the development of many Internet protocols and mechanisms. A first assumption is that (usually) one address is associated to each host. Also, the forwarding of packets is often exclusively based on the destination address. For this reason, there is usually a single path between one source (or client) and one destination (or server). Finally, the Internet was designed with the client-server model in mind assuming that many clients receive information from (a smaller number of) servers. During the last years, these assumptions have been severely challenged. For example, the LISP protocol described in the previous chapters relies on this path diversity.

The client-server model does not correspond to the current operation of many applications. First, large servers are usually replicated and different types of content distribution networks are used to efficiently distribute content [FFM03, Lim, Aka]. Second, the proliferation of peer-to-peer applications implies that most clients also act as servers. This is currently creating several problems in many Internet Service Provider (ISP) networks [KRP05]. Client-server asymmetry does not hold anymore.

Due to the transition from IPv4 to IPv6 many hosts will be dual-stack for the foreseeable future [CLH03]. Furthermore, measurements show that IPv4 and IPv6 do not always provide the same performances, even for a single source-destination pair [ZJUVM07]. This implies that to reach a destination supporting both IPv4 and IPv6, a source host can achieve better performance by selecting the stack that provides the best performance. However, today, this selection is based on simple heuristics. For instance, as highlighted by Matsumoto et al. [MFHK08], IPv6 is chosen prior to IPv4 in most of the dual-stack configurations although IPv4 still remains the best choice from a performance point of view in many environments [ZJUVM07].

We are thus heading towards an Internet that provides a set of potential paths between a source and a destination or a content. Obviously, any path in this set differs from the others as each path has its own performances,

i.e., bandwidth, delay, loss, etc. In such a context, it is important for any application to select their paths in a way meeting their requirements (i.e., not in random way). For instance, bulk data transfer peer-to-peer clients will favor paths with the largest bandwidth so that the targeted file will be downloaded faster. Such a situation is currently discussed within the IETF *Application-Layer Traffic Optimization (ALTO)* Working Group [SB09].

A way to enable efficient path selection for applications would be to allow the network to cooperate with them. Such a cooperation would also give the opportunity to operators for managing incoming and outgoing traffic on their networks. Indeed, according to their traffic engineering needs, operators could balance traffic from one link to another and ensure that some are only used as backup.

In this chapter, we propose a generic informed path selection service called *ISP-Driven Informed Path Selection* (IDIPS). IDIPS is generic as it can be used in many networking contexts without changing anything to its behavior (see Sec. 2.2 for a description of networking use cases). It further does not require fundamental changes in the current Internet architecture and implementation (only the service clients need to implement a library for contacting the service), making its deployment very easy. IDIPS is scalable, lightweight, and designed to be easily deployed in ISP, corporate, or campus networks.

IDIPS is designed as a request/response service. The network operators deploy servers that are configured with policies and that collect routing information (e.g., OSPF/ISIS, BGP) and measurements towards popular destinations. The clients that need to select a path send requests to a server. A request contains a list of sources, a list of destinations and a traffic qualification that determines the rule for ranking the paths to use. The server replies with an ordered list of $< source, destination, rank >$ tuples to the client. The reply gives an indication of the ranking lifetime. This ranking is based on the current network state and policies. The client will then use the first pairs of the list and potentially switch to the next one(s) in case of problems or if it wants to use several paths in parallel.

We implemented IDIPS within the XORP [HHK03] open source routing platform. We describe this implementation in the chapter and discuss cost functions, i.e., functions returning a cost for a given path allowing later ranking. We explain how to construct simple cost functions, such as maximizing the available bandwidth, and demonstrate how to combine them to reflect more complex ranking strategies.

Our evaluation of IDIPS focuses on the whole ranking process, from the request sent by the client to the use of cost function. We demonstrate that our implementation is robust as IDIPS is able to process a large quantity of requests per second while providing a stable response time to the clients.

This chapter is organized as follows: Sec. 2 describes how an informed path selection service should work; Sec. 3 explains how the IDIPS service

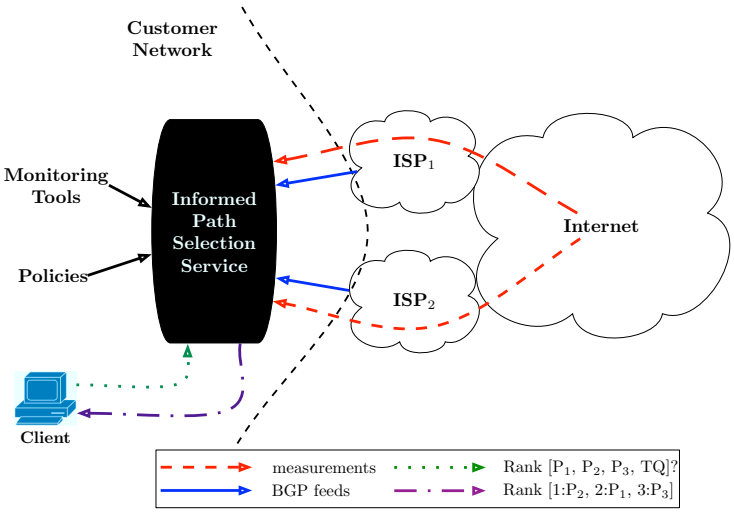



Figure 1: A network service rank paths for the clients

can be implemented; Sec. **??** evaluates our IDIPS implementation; Sec. 6 positions our work regarding the state of the art; finally, Sec. 7 concludes by summarizing our main achievements.

# 2 An Informed Path Selection Service

In this section, we provide a high level description of how should work an informed path selection service. We first explain the behavior of this service (Sec. 2.1) before discussing several use cases that could benefit form the service (Sec. 2.2).

## 2.1 Overview

As illustrated in Fig. 1, an informed path selection service is a request/response service allowing to rank paths. The service is intended to be deployed at strategic points of the network by the network operator inside a domain, a campus, or a corporate network. Clients (see Sec. 2.2 for example of applications that can benefit from such a service) have to implement a library to send requests to the server and use the more preferable paths.

Clients see the service as a black-box. First, a client sends a request to the server (dotted arrow on Fig. 1). This request contains a list of sources and destinations, forming a set of paths to rank, and a traffic qualification (TQ on Fig. 1). The latter argument is a ranking criterion provided by the client. It might be, for instance, “maximize the bandwidth”, “minimize the delay”, or “maximize locality”. The server replies to the request with a

ranked list of paths (e.g., source-destination pairs - dashed-dotted arrow on Fig. 1). The way the ranking is computed by the server remains hidden for the client.

The presence of multiple sources in the request comes from particular situations, such as multihomed hosts (for instance a smartphone with a 3G and a standard wifi connection) or IPv4/v6 dual-stack hosts, where hosts own several addresses.

The server ranks the paths with the help of information from the network:

- Routing information (i.e., BGP, OSPF/ISIS) allowing the informed path selection service to compare different paths based on their routing metrics (e.g., BGP local preference or AS path, IGP cost, etc). This is illustrated in Fig. 1 with the plain arrows.

- Active or passive measurements (i.e., delay, bandwidth, loss, etc) allowing the informed path selection service to compare different paths based on quantitative performance metrics. Note that the server does not necessarily perform the measurements itself. It can request a third party to do the job and retrieve the required performance metrics. This is illustrated with dashed arrows on Fig. 1.

- Policies configured by the network administrator that indicate preferences for some paths over others.

Upon reception of a request, the informed path selection service builds a list of all the possible paths between the source(s) and the destination(s). Then, it removes from consideration the paths that are invalid due to routing (e.g., one destination is not reachable from a given source) or policies. The remaining paths are then ranked according to a set of criteria and the reply sent back to the client contains the following information:

- the best path (source, destination),

- the second best path (source, destination),

- ...

- the $N^{th}$ best path (source, destination)

- the lifetime for the ranked paths.

and for each path its associated rank.

The number of paths returned by the service may be lower, as indicated above, than the total number of possible paths.

Ranking is valid for some time and the client is encouraged to cache the ordered list for the lifetime indicated in the response.

Source and destinations can be IP addresses (either IPv4 or IPv6 or both), IP prefixes, AS numbers, names or of any other type, as long as the client and the server agree on a meaning for them. This possibility to use IP prefixes is motivated by the fact that contiguous IP addresses tend to be used similarly and present similar performances [CH10, SDB09].

IDIPS is designed to be as generic as possible and to prevent an operator to reveal critical information about its topology. This is the reason why the only information returned to the client is the path rank. The rank is an abstraction of information known by the server that defines a partial order among the paths. The order is valid only within one reply and the ranking relationship among any two paths in the reply holds the reflexivity, antisymmetry and transitivity properties.

Path rank is the information revealed to the clients and is computed at the server side with the help of *cost functions*. A cost function determines the cost of using a path. The higher the cost, the less interesting the path. The cost of a path is computed from its *attributes*. The path attributes are the information related to the path. Example of path attributes are its predicted round-trip delay, its reachability or even the AS path from the RIB.

Sec. 2.2 discusses different use cases where IDIPS can be used.

## 2.2 Use Cases

A client of the informed path selection service refers to any entity that has the possibility to select a path to reach a destination or get a content. IDIPS is thus not limited to the user level applications but can also be used directly to improve the routing with the control plane requesting information from IDIPS to make better choices. In this section, we explore several networking scenarios in which the service might find a suitable usage, demonstrating so the general purpose of the service. Additional use cases described in [SB09] can be applied to our informed path selection service.

### 2.2.1 Peer-to-Peer

Peer-to-peer applications are clear candidate users for such an informed path selection service. Each time a peer-to-peer client wants to fetch a given content, it can use the service to rank the various peers sharing the content and, then, select the peer based on the service reply instead of randomly selecting a peer. Ranking, from the client perspective, might be done in order to maximize the throughput, so that the best path will allow a faster download of the content.

Furthermore, not only peer-to-peer clients can benefit from the service but also the ISPs. Indeed, an ISP or a campus network running the informed path selection service could influence providers used by packets sent/received

by hosts of its networks. One can imagine an ISP giving priority to peer-to-peer clients in its own network or in those with who it has a shared-cost peering relationship, avoiding so to pay traffic to its own provider.

The rank is the information abstraction revealed to the client. Internally, an IDIPS server relies on path attributes and cost functions.

Solutions allowing ISPs to help peer-to-peer clients have already been proposed [XYK+08, AFS07]. See Sec. 6 for a discussion on how our informed path service differs from those solutions.

### 2.2.2 IPv4/IPv6 Transition

Due to the transition from IPv4 to IPv6 many hosts will be dual-stack for the foreseeable future [CLH03]. Furthermore, measurements show that, in today's Internet, IPv4 and IPv6 do not provide the same performances, even for a single source-destination pair [ZJUVM07]. This implies that to reach a destination supporting both IPv4 and IPv6, a source can use the informed path service selection to achieve better performance by selecting the stack that provides the best performance.

Today, this selection is based on simple heuristics. For instance, as highlighted by Matsumoto et al. [MFHK08], IPv6 is chosen prior to IPv4 in most of the dual-stack configuration although IPv4 still remains the best choice in most of the environments [ZJUVM07].

### 2.2.3 Multihoming

An increasing number of ISPs, but also campus and corporate networks have chosen to become multihomed by being attached to two or more ISPs [ACK03]. For these networks, multihoming offers two main benefits: technical and economical redundancy, i.e., they remain connected to the Internet even if the link that attaches them to one of their ISPs fails or if one of their ISPs becomes bankrupt. Another important benefit shown by several studies [APS04, AAS03, AMS+08, GDZ06, DD06] is that multihoming allows sites to choose better quality paths over the Internet.

An informed path selection service can find here an usage, for the ISP, in selecting the best provider to reach a given destination/content.

### 2.2.4 Local Network Optimization

The use cases presented above are mostly considering interdomain traffic. Nevertheless, IDIPS can be used for intradomain application. For example, a network optimizing its IGP with OSPF-TE [KKY03] could obtain traffic engineering information regarding its local links via IDIPS. In this case, the IDIPS server would monitor internal path and use information from the IGP to determine the best paths to use. Although this application is promising, we have not investigated more.

### 2.2.5 Traffic Engineering with LISP

LISP, described in Chapter **??**, relies on mappings. A mapping associates a list of locators to an identifier. A priority and a weight are associated to each locator. Only the locators with the lowest priority value can be used, the others can be used only for backup. The weight is used to balance the traffic between the most preferable locators (see Chapter **??**). IDIPS can be used by the mapping owner to determine the priority and the weight of the locators in a mapping. The priorities and weights in LISP offer traffic engineering capabilities to the LISP sites.

# 3 Service Construction

In this section, we describe how we implement an informed path selection service. Our implementation is called *ISP-Driven Informed Path Selection* (IDIPS) and it architecture is described in Sec. 3.1. We further discuss how our implementation is included in the XORP framework [HHK03] (Sec. 3.2). Finally, we explain in details how to build simple cost functions and combine them to reflect more complex ranking strategies (Sec. 3.3).

## 3.1 Architecture

IDIPS is composed of three independent modules: the *Querying* module, the *Prediction* module, and the *Measurement* module. The Querying module is directly in relation with the client as it is in charge of receiving the requests, computing the path ranking based on traffic qualification provided by the client and the ISP traffic engineering requirements, and replying with the ranked paths. For the sake of generality, the remainder of this chapter will use the term *ranking criterion* when referring to traffic qualification. The Measurement module is in charge of measuring path performance metrics if required. Finally, the Prediction module is used to predicting paths performance (i.e., future performance metrics of a given path based on the past measurements).

The ranking criterion provided by clients in their requests might require measuring the network to obtain path performance metrics, such as delay or bandwidth estimation. One of the key advantages of IDIPS is that it avoids clients measuring themselves the network, leading to redundant traffic injected in the network (See Chapter **??**). The Measurement module performs the measurements or asks a third-party to perform the measurements. Those measurements can be active (i.e., probes are sent in the network) or passive (i.e., no additional traffic is injected).

It is possible to predict the performance of a given path if it has been previously measured [YRCR04, DCKM04, PLMS06, Pap07, dLUB05, WSS05, LPS06, LGS07, LHC03, LGP+05, NZ04, FJP+99, NZ02, PCW+03, ST03,

LHC05, CCRK04, RMK+08, MS04]. This prediction task is achieved by the Prediction module. Note that a given measurement can be used in several different predictions. For instance, the previous delay measurements can serve for predicting the delay, the jitter, or for determining whether the path is reachable or not.

To enable flexibility, ease of implementation and performance[1], IDIPS clearly separates the Querying, Measurement and Prediction modules. Each instance module communicating with the other modules thanks to a standardized interface. Therefore, the handling of requests from the clients is strictly separated from the prediction of path performance and path performance prediction is separated from path measurements.

The Querying module receives the ranking requests from the clients and computes the rank for these requested paths based on their predicted future performance. Future paths performance are estimated by the Prediction module that relies on the measurements performed by the Measurement modules.

All along this chapter, we are using the terms measurements and predictions however, they have to be understood in their very generic meaning. For IDIPS, a measurement corresponds to any information grabbed from the network. This definition encompasses active measurements like pings, passive measurements like Netflow information [Cla04] or even routing information like BGP feeds. Likewise, a prediction in IDIPS is an information that is likely to be valid in the coming future. Therefore, a prediction can be the result of very complex machine learning techniques but also very simple information like the originating AS of the path destination. In other word, a measurement is an information discovered in the past and a prediction is an information that is likely to be valid in the coming future.

To support as many requests per second as possible, the IDIPS modules are running independently of each others. This independence is ensured through the use of caches. Each module stores its processing results in its local cache. If another module requests a given result, a simple `get` in the appropriate cache will return it.

There may exist several instances of the Prediction and Measurement modules. For example, IDIPS can have a delay measurement module, a bandwidth measurement module, a delay prediction module, and a bandwidth prediction module. Sec. 3.2.1 gives an example of Measurement and Prediction modules implementation.

### 3.1.1 Querying module

Common applications are only able to use one path at a time, even if several exist. In this case, the client only needs to know the very best

[1] IDIPS must potentially handle many ranking requests simultaneously

path returned by IDIPS when it has no additional information about the paths. For this reason, the list of ranked path is sorted by rank before being transmitted to the client. Then, the client can safely consider the first path of the list as the very best path (or one best path among all the best paths if several ones have the same lowest rank value). The other paths are returned only for resiliency (the best path is not valid for the client) or if the client uses the ranked list to refine a local decision. Sorting the paths simplify the operation at the client.

Paths ranking is done with the use of *Cost Function*. For a given <source, destination> pair, the cost function returns a cost, i.e., a positive integer resulting from metrics combination of a given path. The lower the path cost, the more attractive the path. By definition, the sum of several costs is also a cost. It is therefore possible to combine cost functions with a weighted sum in order to reflect complex strategies or politics. Sec. 3.3 explains how to construct cost functions.

To support as many requests per second as possible, the IDIPS modules are running independently of each others. This means that the Querying module never has to wait for a path performance prediction to be computed by the Prediction module to compute the path ranking. When a prediction has to be retrieved by the Querying module, it calls a `get` on the Prediction module for the path attribute it is interested in. The attributes of a path are the predicted metric values as computed by the Prediction module for the path. For the sake of generality, any attribute is encoded as an integer. If an information is too complex to be represented with a single integer, it can always be represented as a set of integers. For example, an $< x, y >$ coordinates can be decomposed in the `x_coordinate` and the `y_coordinate` and a function that needs to use the coordinates just needs to retrieve the `x_coordinate` and the `y_coordinate` to reconstruct the full coordinates. Sec. 3.1.3 gives more details about the interface to retrieve path attributes from the Prediction module.

Depending on its needs, a client can query IDIPS in a *synchronous* or *asynchronous* way. In synchronous mode, when a request is received by the IDIPS server, the server sends the list of ranked paths back to the client once computed. On the contrary, in asynchronous mode, when a request is received by the IDIPS server, the server computes the paths ranking but does not send the list back to the client. The requester must explicitly send a special command to retrieve the list of ranked paths. The API that IDIPS presents to clients is depicted in Fig. 2 for the synchronous mode and in Fig. 3 for the asynchronous mode.

The commands are sent by the client to the server. When the client uses the asynchronous mode, it receives a transaction identifier (*tid*) back from the server. Every request received by a server is abstracted as a transaction. This *tid* is the identifier of that transaction on the server. This identifier is used for retrieving the list of ranked path with the `get_all_path_ranks`. If

```
sync_rank_paths
         ? sources & destinations & criterion
         -> ranked_paths_list & ttl
```

Figure 2: IDIPS server API for synchronous mode clients

```
async_rank_paths
         ? sources & destinations & criterion
         -> tid

get_all_path_ranks
         ? tid
         -> ranked_paths_list & ttl

get_next_path_rank
         ? tid
         -> source & destination & rank & ttl & more

get_next_n_path_ranks
         ? tid & n
         -> ranked_paths_list & ttl & more

terminate_transaction
         ? tid
```

Figure 3: IDIPS server API for asynchronous mode clients

the ranking is not yet computed by the server when the `get_all_path_ranks` is received, an empty list of ranked paths and the invalid `0x0` ttl are returned. The server, in asynchronous mode, always returns immediately a result when it receives an `async_rank_paths` or a `get_all_path_ranks`. The client must then poll the server until it has retrieved the list. This behavior is used to avoid the server to maintain state about the clients, it only maintains ranking state. To avoid the need of client polling, signaling could be used to let the server inform the client that the transaction is ready but it thus means that the server must maintain state about the client, which is what we want to avoid while using asynchronous mode. Polling is by definition avoided in synchronous mode. It is worth to notice that a ranking call can be implemented as being blocking or non-blocking at the client side, independently of the client to server communication mode. The typical use of a blocking call is when the path to exchange data cannot be changed once the flow is started. Then, the best path must be used. The client must then wait for the path ranking before being able to exchange data. On the contrary, non-blocking call is used when the client can change the path it uses while exchanging data. For example, a shim6 [NB09] host starts exchanging data with a path arbitrarily selected by following the rules of RFC3484 [Dra03]. If the data transfer is long enough, shim6 could decide to switch to the best path computed by IDIPS. In this case, the flow can start as soon as possible, even if the path used to exchange data might be sub-optimal at the beginning.

To avoid this waste of resources, IDIPS also offers the possibility to retrieve one path at a time with the `get_next_path_rank` that returns the best path that has not yet been retrieved by the client. To use the best working path, the client can use the algorithm presented in Fig. 4 where `handle_path` is the client function that needs the path and that returns true when no more path is needed. The `more` parameter returned by the `get_next_path_rank` indicates if there is still a path to retrieve for the transaction. Optionally, the client can explicitly ask IDIPS to terminate the transaction. If not, IDIPS should eventually terminate it automatically. Instead of considering retrieving the rankings one by one or all at a time, the more generic `get_next_n_path_ranks` is also proposed where the client specifies the number of paths that must be returned by IDIPS. The equivalent of GET_ALL_PATH_RANKS corresponding to a specified number higher or equal to $\left(\overline{\texttt{sources}} * \overline{\texttt{destinations}}\right)$ while a value equal to one corresponds to the `get_next_path_rank`. However, in most of the cases, a client is interested by either one or all the paths.

We suggest to use UDP to exchange message between the clients and the servers. Using UDP avoids the burden of establishing and maintaining TCP connections. However, IDIPS does not preclude the use of another protocol or even several protocols at the same time. For example, a server can be requestable via UDP by lightweight clients and propose a more complex

```
more := true

WHILE more
DO
   (src, dst, rank, more) := get_next_path_rank(tid)
   IF handle_path(srcs, dst, rank)
   THEN
       STOP
   END
DONE

terminate_transaction(tid)
```

Figure 4: One-by-one path ranking retrieval algorithm

```
start_measurement ?  source & destination & interval

stop_measurement  ?  source & destination

set_interval      ?  source & destination & interval

get_measurements  ?  source & destination
                  -> measurements
```

Figure 5: Measurement module API

interface via HTTP/XML for more powerful clients.

### 3.1.2 Measurement Module

The Measurement module is in charge of measuring the paths. The measurements can be active or passive. For example, an active measurement could be a ping and a passive measurement could be the count of the number of TCP SYNs entering the network.

The Measurement module API presented in Fig. 5 is two fold. The `start_measurement`, `stop_measurement` and `set_interval` commands determine the targets to measure while the `get_measurements` is used to retrieve the last measurements of a path.

Measurements are always defined between a source and a destination and are performed periodically (with a configurable interval between the measurements). The possibility to modify the interval of a measurement is not mandatory but is more convenient as it allows one to adapt the mea-

```
start_prediction ?  path

stop_prediction  ?  path

get_prediction   ?  path
                 -> prediction
```

Figure 6: Prediction module API

surement rate dynamically without disrupting a measurement campaign. If such command is not available, it means that the measured values must be stored outside the measurement module. Indeed, without the `set_interval` command, the measurement has to be stopped, then re-started from scratch meaning that all the state in the measurement module instance is lost for this measurement. Finally, the `get_measurements` command returns all the measurements performed so far for the <source, destination>.

The decision of measuring a path is done either by configuration or triggered by the Prediction module.

### 3.1.3 Prediction Module

The prediction module contains all the intelligence of IDIPS. Indeed, IDIPS is a service that aims at determining the best paths to use. However, determining the best path to use is a prediction exercise as the future behavior of a path is seldom know, particularly when considering inter-domain paths. Determining how to predict a path behavior the best is out of the scope of this chapter but this section presents how a Prediction module has to be implemented in IDIPS.

As already expressed earlier, IDIPS modules are running independently. However, the Querying module needs to know the path attributes computed by the Prediction module. In addition, the Prediction module has to know the path it has to predict the performance metric for. To this aim, the Prediction module provides the API presented in Fig. 6.

This API has two components. On the one hand, the `start_prediction` and `stop_prediction` commands are used to specify the path to predict performance metric for. On the other hand, the `get_prediction` command is used to retrieve the predictions.

`get_prediction` always returns a value. If the attribute value is not defined, an error or a meaningful default value is returned. For example, if the bandwidth of a path is not known a default value of zero can be returned making the path less interesting than any other path.

The decision of measuring or predicting a path is highly related to the deployment policies, the topology and the traffic. The decision of predicting a

path is thus not provided by IDIPS but is considered case by case by the Prediction module or by configuration. There exists three ways of determining if a prediction has to be started or stopped. First, an operator can manually determine the path to do prediction for and uses `start_prediction` and `stop_prediction` commands to do so. Second, a prediction module instance can determine by itself if a path is worth being measured or not. For example, if a prediction module received enough `get_prediction` for a path it is not predicting yet, it can decide to start predicting it. In this second case, the `start_prediction` and `stop_prediction` commands are not used. Finally, a prediction module instance can predict that a path has to be predicted and command another prediction module to start predicting the path. For example, a prediction module instance can be in charge of predicting if a path is important or not based on the traffic it carries. If the path is considered as important, it can ask to start the delay prediction for that particular important path.

To predict the future path behavior, a prediction module often needs information from the measurement module. Like the Querying module can retrieve a prediction with a simple `get`, the Prediction module can retrieve the measurements from the Measurement module with the `get_measurements` (see Sec. 3.1.2). The Prediction module can use the last measurements to predict the future behavior of a path. Based on the prediction and on its quality, the prediction module can decide to modify the frequency at which a measurement has to be performed (with the `sec_interval` command) or ultimately to start or stop a measurement. In addition, because a prediction module aims at providing the path performance for the near future, the `get_prediction` only returns one result as opposed to `get_measurements` that returns a list of measurements. Obviously, this API does not preclude an extended API that would return more information about the quality of the prediction (for example a TTL) or several predictions at once.

## 3.2 XORP Implementation

Sec. 3.1 presents a potential generic architecture for IDIPS. In this section, we present how we have implemented IDIPS within the XORP framework [HHK03]. XORP is an open source routing plateform that facilitates the implementation of control plane protocols. In XORP, each control plane protocol (this including their communication channels) is implemented as an independent process. The XORP inter-process communication is ensured by the use of the *Xorp Resource Locators (XRL)*. An XRL is very similar to RPC but much simpler to implement and use. XRLs are always asynchronous. It means that once a process has sent an XRL, it does not block waiting the answer. When the target of the XRL has computed the result, it will instead trigger the call of a callback function at the process that sent the XRL. This is possible because a callback function (a C++ method ref-

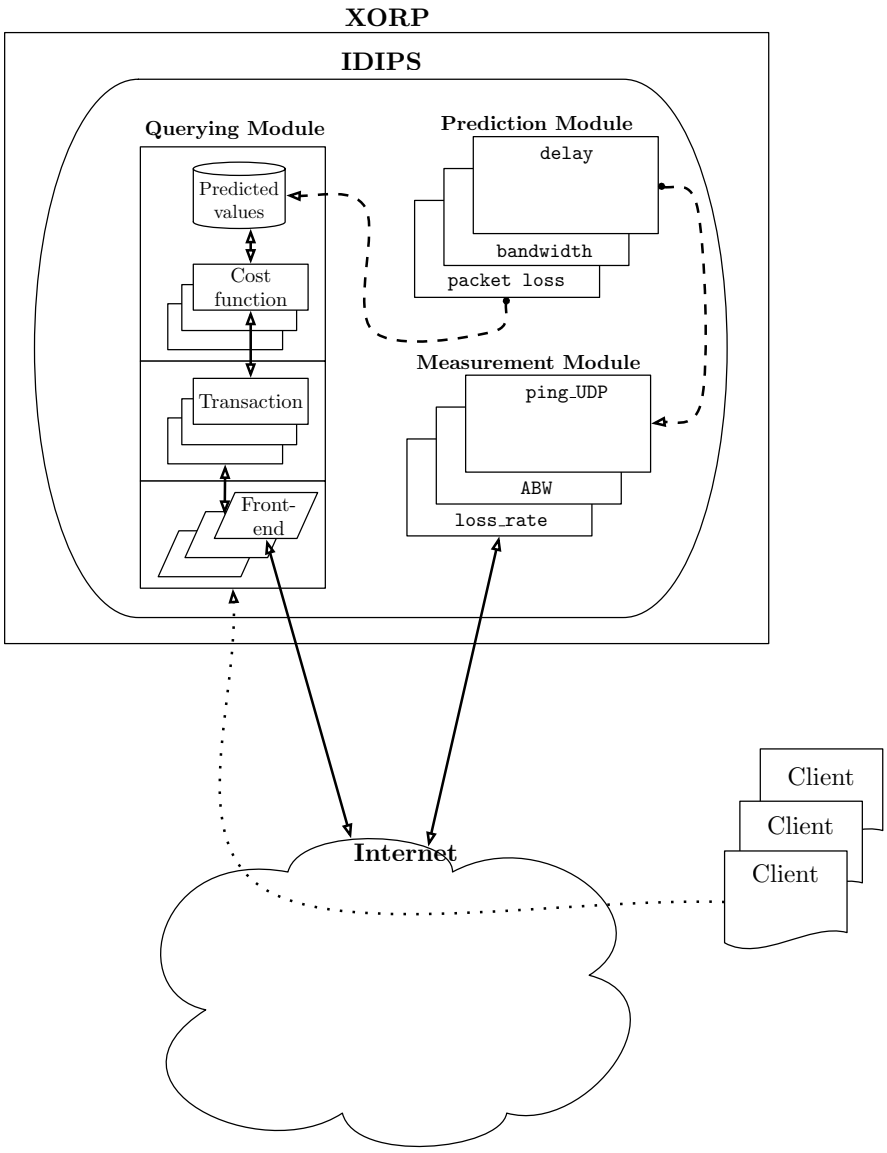



Figure 7: IDIPS within XORP

erence) is always associated to an XRL call. The callback is just a call to the method on the instance with the parameters set to the values computed by the target of the XRL. We have chosen to implement IDIPS within XORP for two reasons. First, implementing IDIPS in XORP gives direct access to the routing table, meaning that routing information can be easily injected into IDIPS. XORP provides implementations for the most common control plane protocols, e.g., BGP, with well defined XRL interfaces to communicate

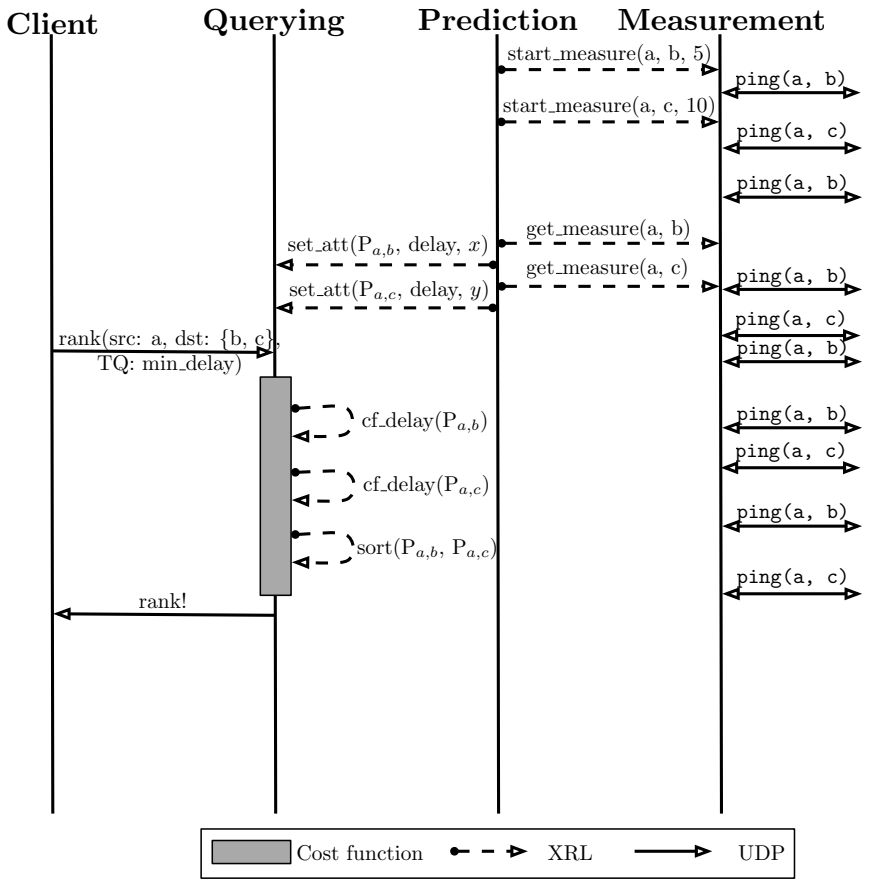



Figure 8: Example of modules interactions in IDIPS

with. Second, XORP is designed to be distributed. It means that it is possible to have processes interacting together but running on different hosts. We do not use process distribution in our prototype however, one could imagine running the prediction modules on dedicated servers or distribute the measurement module.

As in the generic architecture presented in Sec. 3.1, our implementation is decomposed in the Querying, Prediction and Measurement modules. The querying module is a single XORP process, while there are as many XORP processes as required to implement the Measurement and Prediction modules. For example, if IDIPS requires to measure the delay and the bandwidth, the Measurement module will contain two XORP module. One implementing a delay measurement and the other implementing the bandwidth measurement. Fig. 7 shows the IDIPS architecture in XORP, while Fig. 8 shows how the different modules interact with each others.

The Querying module is decomposed in three main parts: (*i*) the *Front-end* part, (*ii*) the *Transaction* part and (*iii*) the *Cost function* part. The Transaction part receives the requests from clients and returns the ranking results. The Transaction part processes the requests received by the Front-end and computes the path rank for these requests. Finally, the Cost function part implements the cost functions. Our implementation has two different Front-ends. On the one hand, a client can request IDIPS to rank paths by sending UDP messages. On the other hand, any XORP process can request paths ranking just by sending an XRL to the Querying module. The typical use of UDP messages is for clients independent of IDIPS, e.g., a P2P client while the XRL interface allows any XORP process to use IDIPS to improve its decision. For example, the FIB as computed by XORP could be optimized by taking IDIPS ranks into account.

IDIPS must potentially handle many ranking requests at the same time. To support a potentially high load, requests are abstracted into *transactions*. Therefore, for each request, a transaction instance is created by a unique identifier. Each transaction runs independently of the others and maintains the list of sources, the list of destinations and the path ranking criterion. If the request uses the synchronous mode, the transaction also maintains information to send the reply to the requester. When a request is received, the Front-end instantiates an empty transaction and adds all the paths from the request. At that stage, the paths are computed blindly: for each source $s$, for each destination $d$ in the source and destination lists, the $< s, d >$ path is added to the transaction. Once all the paths are added to the transaction, the `run` method is called on the instance.

The job of the `run` method is to determine the cost and the rank of each path, according to the path ranking criterion and to build the sorted list of ranked paths. A transaction is *ready* once the ordered list of ranked paths has been built completely. The cost of each path is determined by calling the appropriated cost function on the path. Once the cost is determined for a path, the path, tagged with its cost is added to the priority queue `_costs`. The `_costs` structure is maintained ordered by the path cost. It means that at any time, the $i$th entry in `_costs` has a lower or equal cost than the $i+1$th entry. The transaction is set ready once the cost and rank of each path is known. If the request was in synchronous mode, the transaction triggers the transmission of the reply to the request once the transaction becomes ready. If the request was in asynchronous mode, the method stops. As long as the transaction is not ready, a call to retrieve a path for an asynchronous mode request returns an error.

We have implemented the call of the cost function in two different ways. By using XRL or by directly calling the method on the querying module class instance. We use the XRLs to parallelize the processing. However, as the processing of XRL is centralized (via the `finder`) and because the management of XRLs is sequential and implemented with a list, this im-

plementation does not improve the performance, even worse, it reduces the number of requests IDIPS is able to sustain and may cause ranking failures because XRLs can be lost. Indeed, the XRLs are enqueued in a list limited in size and number of entries. Therefore, once the list is full, XRLs can be lost. The performance can also drop because an XRL at position $i$ in the queue will not be dispatched to the Querying module before the XRLs prior to position $i$ have been dispatched to their target process. Even if the processes are different. With an experiment where the requests ask to rank 50 different paths, we observed a drop of 54% of requests per second supported by IDIPS compared to an implementation calling the cost function directly without XRLs. We also noticed about 12% of failing transactions and a time to compute the rank 56 times higher with the XRL implementation. However, for the requests that succeeded, the time perceived by the client was 13% faster with the XRL implementation. The time perceived by the client is the time elapsed between the sending of the request and the reception of the ordered list of ranked paths. Despite the better client perceived time with the XRL implementation, we recommend not to use the XRL implementation. Indeed, without the use of XRLs, IDIPS can handle more simultaneous requests and does not face loss of requests due to the limited size of the XRL queue.

Sec. 3.1 proposes to keep the modules independent thanks to the use of getter functions: when a module needs information from another module, it sends a get to the module to retrieve the values. In our implementation, every module implements such getters. However, we have also implemented a paths attributes cache within the Querying module. This cache stores, for each path, all the known attributes for the paths. The attribute values are computed by the Prediction module. This cache is based on a *push* model. It means that it is not the Querying module that populates it but the Prediction module that pushes the values to that cache. The querying module thus implements the `set_attribute` and `get_attribute` XRLs. Therefore, when a prediction is computed, the Prediction module immediately calls the `set_attribute` XRL on the Querying module to set the attribute value for the path that as just been computed. This mechanism is implemented to speed-up the cost computation for the paths. Indeed, as presented in Sec. 3.3, the cost of a path is computed with a cost function that potentially needs the attributes of the path. Thus, without an attribute cache at the Querying module, an XRL must be called to the appropriate Prediction module instance for each attribute to retrieve. However, calling XRL implies some delay as the call must be sent first to the main XORP process, i.e., the finder, then it is sent to the appropriate Prediction module instance. This delay can become non negligible if the Prediction module is not running on the same host as the Querying module. For this reason, the Querying module does only rely on this cache. If the cache has no entry for the path attribute, it is considered that the path is not under measurement/prediction

and the Cost function must determine an appropriate cost. It is important to remark that our implementation does not allow the prediction module to determine by itself that a path merits to be predicted. Indeed, the Querying module never calls the `get_attribute` on the prediction module so the prediction module cannot count the number of failing calls. However, one could imagine a measurement module instance monitoring the cache misses at the querying module. The prediction module could then determine the paths that are worth being predicted.

The notion of module is translated into XRL interfaces in XORP. Except for the Querying module, there might be several C++ classes implementing a module and possibly several instances of a class as illustrated in Fig. 7. Each class must implement the XRL interface corresponding to the module it is related to. Fig. 9 gives the XRLs that must be implemented by the class implementing the Querying module. It is important to notice that the interface for the querying module is only composed of the setter and the getter for the path attributes. It does not include an interface for clients to query IDIPS. Indeed, XRL interfaces are only related to the implementation. Nevertheless, we have implemented the client-related commands described in Fig. 2 and Fig. 3 with XRLs to make IDIPS usable directly by any XORP process. It means that our IDIPS implementation has two front-ends, one listening on UDP and the other listening on XRLs. Fig. 10 lists the XRLs that must be implemented by the classes implementing the Measurement module. Finally, Fig. 11 shows the XRLs that the classes implementing the Prediction module must implement. Each class implementing one and only one technique. For example, one class can implement a UDP ping for the measurement module and another can measure the path bandwidth and one class can implement a delay bandwidth product predictor based on the delay and bandwidth measurements.

The whole process is presented in Fig. 8. The Prediction module asks an instance of the Measurement module (i.e., the delay measurement instance) to measure a path. A path to measure is defined by a source and a destination. For the sake of generality, the source and the endpoint of any path to measure is represented textually, meaning that it can be a name, an IP address a network interface, or any other suitable information. Each path installed in a measurement module is periodically measured with a configurable interval between measurements (e.g., 5 for path (a,b) and 10 for (a,c) in Fig. 8). The use of IP prefixes instead of IP addresses is particularly interesting to aggregate information. For example, if a site has one IP prefix $p/P$ for its clients and that the performance are considered to be the same for any of them, then all the paths can be aggregated by using the $p/P$ source instead of the client IP address.

The `start_measurement` XRL function triggers the measurement of the path defined by the `source` and `destination` parameters. The path is then

```
interface idips_querying/0.1 {
  /**
   * Get a path attribute
   * @param path to get the attribute from
   * @param name of the attribute
   * @param value of the path attribute
   * @param rpath echo of path
   */
  get_attribute?path:txt&name:txt->value:u32&rpath:txt;

  /**
   * Set a path attribute
   * @param path to set the attribute to
   * @param name of the attribute
   * @param value of the path attribute
   */
  set_attribute?path:txt&name:txt&value:u32;
}
```

Figure 9: XORP Querying module XRL interface

measured every `interval` seconds (e.g., 5 for the path (a,b) in Fig. 8).[2]

The various Measurements module instances keep locally the last measurements they obtained for the paths they are measuring. When a Prediction module needs a measurement, it sends a `get_measurement` XRL to the adequate instance of the Measurement module and retrieves the measurements for the path. The measurement is then sent to the Querying module, with the `set_attribute` function, for being stored in the Predicted values storage.

### 3.2.1 Examples of module implementation

This section presents two examples of module implementation. We first present a Measurement module that implements a UDP ping and then describe a Prediction module that implements an average delay predictor. The Prediction module uses the measurement module to predict the delay of the paths.

**Measurement module example** Our Measurement module example implements a simple UDP ping. To estimate the round-trip delay between a <source, destination> IP pair, we send a UDP segment to the destination on a port number that is very unlikely to be open. If the port is not opened

[2]To avoid synchronization, the time between two measurements should be set to be equal to the `interval` parameter on average.

```
interface idips_measurement/0.1 {
  /**
   * Start periodically measuring a destination
   * @param destination destination to measure
   * @param interval interval in seconds between two measurements
   */
  start_measurement?source:txt&destination:txt&interval:u32;

  /**
   * Stop measuring a destination
   * @param destination destination to stop measuring
   */
  stop_measurement?source:txt&destination:txt;

  /**
   * Change measurement interval for a destination
   * @param destination destination to change the measurement interval
   * @param interval new measurement interval for the destination
   */
  set_interval?source:txt&destination:txt&interval:u32;

  /**
   * @params destination  destination to get the past measurements
   * @params measurements list of measurements
   * @params clean remove elements after retrieving them
   */
  get_measurements?source:txt&destination:txt&clean:bool->measurements:list<u32>;
}
```

Figure 10: XORP Measurement module XRL interface

```
interface idips_prediction/0.1 {
  /**
   * Start a prediction model for a path
   * @param path to predict
   * @param src source IP for the measurements
   * @param dst destination IP for the measurements
   */
  start_prediction?path:txt&src:ipv4&dst:ipv4;

  /**
   * Stop a prediction model for a path
   * @param path to stop the prediction for
   */
  stop_prediction?path:txt;

  /**
   * Get the prediction for a path
   * @param path to get the prediction for
   * @param prediction for the path
   */
  get_prediction?path:txt->prediction:u32;
}
```

Figure 11: XORP Prediction module XRL interface

and if no filtering applies, an ICMP port unreachable is expected to be returned to the Measurement module. The sending of the UDP segments is done by using the XORP socket API. XORP sockets are similar to the POSIX sockets except that they are asynchronous and that they are implemented with XRLs. In the reminder of this section we will use the term *socket* to refer to the XORP socket abstraction. A XORP process that wants to use a socket has to implement the `socket4_user`[3] XRL interface. This interface defines several XRL like `error_event` or `recv_event` that respectively indicate if an error occurred with the socket or if bytes are ready to be read on a socket. The `socket4_user` is used to signal the XORP process about events on the sockets it is in charge of. To open, bind, connect, listen, send data on or close a socket, the IDIPS must use an XRL Socket Client. XRL Sockets Clients are classes that implement the `socket4` XRL interface and are directly provided in the XORP framework.

To implement the UDP ping, we create one connected UDP socket per <source, destination> IP pair and periodically send a UDP socket with it. The time at which the packet is sent is stored for later use. Because the destination does not listen on the port, it sends an ICMP port unreachable that eventually triggers the call of the `error_event` XRL in our process.

[3] `socket6_user` for IPv6

The error indicates on which socket the error arrives and the nature of the error. The delay is thus simply computed by doing $now - measure$ where $now$ is the time at which the XRL is called and $measure$ is the time at which the probe was sent.

The module needs to keep some state about the <source,destination> IP pairs it measures. To do so, different datastructure are required. First, the `_destinations` map maintains measurement information for each <source, destination> IP pair. This information contains the interval at which the pair must be measured and the list of the measured delays for the pair (the closer to the end of the list, the more recent the measurement). Once a delay has been measured for a pair, it is appended to its measured delay list. When the `get_measurements` command is called on the measurement module, this is the measured delay list for the requested pair that is returned. Two other datastructures are used to map a socket identifier to a pair and vice versa. The `_socket_info` maps gives information about the socket indexed by the socket identifier. The related information is the source and destination addresses and the time at which the last segment has been sent on this socket (the *measure* variable). The `_sockets` map is the opposite of the `_socket_info`. `_sockets` gives the socket identifier for any pair. The `_socket_info` is unfortunately required as there exists no way in XORP to retrieve meta information on a socket like those we need.

The IP pairs are measured periodically. To implement these periodic probings, we use a XORP periodic timer. Every second, this timer calls the `loop` method of our process. When this method is called, a UDP segment is sent to each <source, destination> IP pair that should have been measured at the latest when the `loop` method is called. To efficiently determine the pairs to measure at the `loop` call, the `_to_measure` priority queue is maintained for each source covered by the measurement module. The key in the priority queue is the time at which the measurement has to be done and the value is the destination address. When a measurement is sent by the `loop`, the entry is removed from the priority queue and the next measurement time is computed for that entry. The new measurement time is then added to the priority queue. Fig. 12 shows the pseudo-code of the `loop` method.

Lines 17 – 21 ensure the measurement periodicity of <source, destination> pair that has not been stopped. The salt is used to avoid synchronization of measurements and is a small random value [AKZ99].[4] With Fig. 12, we can see that stopping a measurement by calling the `stop_measurement` does not apply immediately and an ultimate probe is sent after such a call. We can also see that there is never more that one entry par <source, destination> pair in the queue which is optimal from a memory point of view.

Fig. 13 shows how the ICMP port unreachable is processed by our module.

[4] In our implementation, the salt is zero.

```
00 FOREACH source IN _to_measure
01 DO
02      WHILE _to_measure[source] IS NOT EMTPY
03      DO
04          entry := _to_measure[source].pop
05
06          IF entry.key > NOW
07          THEN
08             MOVE TO NEXT SOURCE
09          END
10
11          destination := entry.address
12          socketid := _sockets[source][destination]
13          _socket_info[socketid].last_call := NOW
14
15          send_UDP_probe(socketid, source, destination)
16
17          IF (source, destination) NOT STOPPED
18          THEN
19              entry.key := NOW + _destinations[source][destination].interval
                                        + salt
20              _to_measure[source].push(entry)
21          END
22      DONE
23 DONE
```

Figure 12: Measurement module `loop` method pseud-code

```
SOCKET4_USER_0_1_ERROR_EVENT(socketid, error)
00 now := NOW
01 IF error = ICMP_PORT_UNREACHABLE
02 THEN
03      si := _socket_info[socketid]
04      measure := si.last_call
05      delay := now - measure
06      _destinations[si.source][si.destination].measurements.append(delay)
07 END
```

Figure 13: UDP ICMP port unreachable management

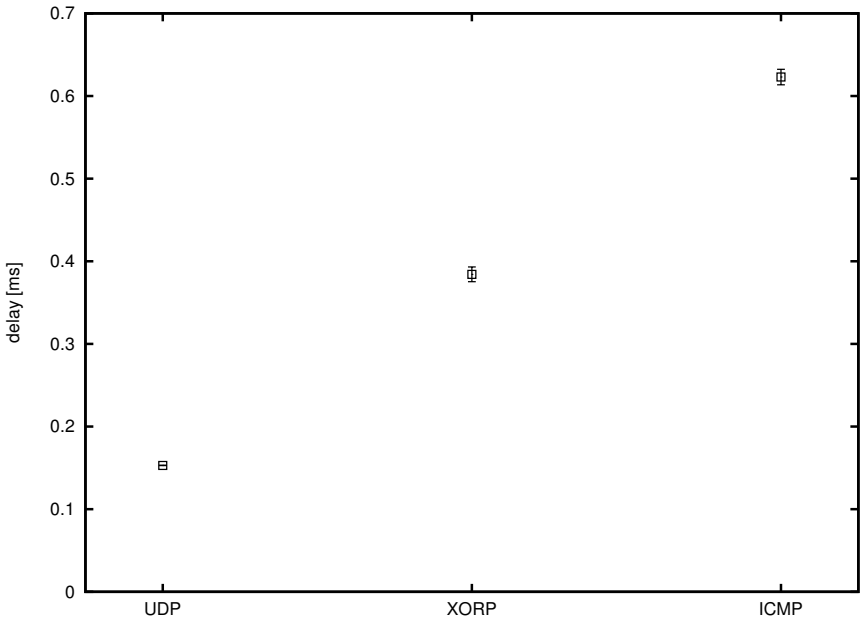



Figure 14: Delay precision comparison between standard UDP ping, ICMP ping and XORP UDP ping

It is not possible, without changing XORP to associate a time to an event on a socket. This explains why line 0 is required in the algorithm of Fig. 13. The retrieval of the time has to be carried out as soon as possible to limit the inaccuracy of the delay estimation.

Fig. 14 compares the accuracy of the UDP ping we have implemented in a Measurement module with the standard traceroute UDP ping and the ICMP ping command. Fig. 14 plots the average of delay measured by the technique and the 95% confidence interval. For the comparison, we made 1,500 pings for each technique. The interval between two probes is one second. The setup uses two machines directly on the same VLAN. One running IDIPS on Linux, and the other receiving the UDP probes and running on Linux. The UDP ping is performed with the `traceroute` command and the ICMP ping with the `ping` command.

From Fig. 14 `UDP ping` and `XORP` labels, we can see that using XORP introduces a bias in the measurement. This bias is mostly due to the process context switching introduced by the XRL based implementation of the XORP sockets. Indeed, when an segment arrives at the host, it is not delivered immediately to the measurement process. It is first received by the main XORP process that then notifies the Measurement module process that an event occurred on the socket (i.e., the reception of the ICMP port unreachable). This design thus imply process switching in addition to the

standard kernel/user space switching with POSIX sockets. In addition, this scheme is also applied when sending the segment meaning that the time between the probe is sent from the Measurement module process and the time the probe is effectively received by the kernel depends on how rapidly the XRL is sent and processed by the main XORP process. However, the bias is of less than 0.3ms in our setup which is acceptable for most of the measurements. More interestingly, the measurement is better with the UDP ping implemented in IDIPS than the standard ICMP ping.

**Prediction module example** The Measurement module presented above does delay measurement by the mean of UDP pings. The Prediction module example in this section uses the round-trip-delays measured by the UDP ping measurement module to predict the delay expected for the paths in the near future. The Prediction module simply averages the last round-trip-delays measured for a <source, destination> IP pairs. The average delay is the prediction of the delay for the path defined by the pair.

In this module, a path is defined by a source and a destination IP address. When a `start_prediction` command is received by the prediction module, it requests the UDP ping measurement module to start a measurement for the <source, destination> IP pair that defines the path the delay prediction has to be performed for. The prediction module then periodically retrieves the list of the last measurements for the path. Because the prediction module is the single one to use the UDP ping prediction module, it requests the measurement module to flush its memory. The prediction module then computes the average of the measured delays in the list. This average is considered as the future value of the delay until the next retrieval of the measurements list for the path.

The prediction module maintains two datastructures. On the one hand, the `_paths` map maps a path to a <source, destination> IP pair. On the other hand, the `_delays` map stores the predicted delay for each path.

To speed-up the Querying module processing, the prediction module also pushes the prediction delays to the Querying module path attributes collection. That is, when the Querying module needs the delay prediction, it does not need to request the prediction module. Doing so limits the use of XRLs and thus the number of context switches.

Our example implementation has no other intelligence. Indeed, the list of measurements is retrieved at the same rate for each path (once every 10 seconds thanks to a XORP periodic timer) and the prediction module requests the UDP ping measurement module to send a probe every second. However, it would not be a hard task to modify the module to enable an adaptive measurement rate and an adaptive measurements list retrieval.

**Algorithm 1** Example of cost function for the reachability

**Ensure:** Integer value representing the result of this Cost Function.

```
1: procedure IS_REACHABLE_CF(src, dst)
2:     reachable ← get_attribute(<src,dst>, REACHABILITY)
3:     return reachable
4: end procedure
```

### 3.3 Cost Functions Implementation

In this section, we show how to construct simple fundamental cost functions and how to combine them to implement an ISP policy. Our example is based on a situation in which an ISP has three customer families: (*i*) *premium users* always requiring the best available performances, (*ii*) *standard users* requiring a good performance/cost trade off, and (*iii*) *light users* always requiring the lowest cost. The traffic engineering changes between the night and the day for standard users: during the day, a lower cost is preferred while during the night, the performance is preferred. The monetary cost of a path depends on the $95^{\text{th}}$ percentile load of the link used to reach the Internet.

In our example, we assume that the prediction module feeds the querying module with the following information:

- routing reachability of the paths. A path is reachable if there exists a route in the FIB to forward traffic from its source to its destination, this information is stored in the `REACHABILITY` attribute
- originating ASN. The originating Autonomous System Number of a path is the originating AS number of the prefix of the destination as discovered by BGP. This information is stored in the `ORIGIN` attribute
- monetary cost of the paths. The monetary cost of a path is the expected cost it would represent to carrying one additional Mega bit per second of traffic on it. This cost is computed by applying the $95^{\text{th}}$ percentile technique [DHKS09] and is stored in the `COST` attribute
- available bandwidth of the paths. The available bandwidth of each path is estimated and is expressed in Mbps stored in the `ABW` attribute
- customer family. A customer can be premium, standard or light user. The customer family, stored in the `FAMILY` attribute, of a path is determined simply by considering the source of the path and ignoring its destination

We first have to define if a destination is reachable or not from a given source address. A path, defined by a <source, destination> pair, has its

**Algorithm 2** Example of cost function for the path locality

**Ensure:** Integer value representing the result of this Cost Function.

```
1: procedure LOCALITY_CF(src, dst)
2:     origin ← get_attribute(<src,dst>, ORIGIN)
3:     if origin = LOCAL_ASN then
4:         return 0
5:     end if
6:     return 1
7: end procedure
```

**Algorithm 3** Example of cost function for the cost minimization

**Ensure:** Integer value representing the cost of using the path defined by *src*, *dst*.

```
1: procedure MINIMIZE_COST_CF(src, dst)
2:     cost ← get_attribute(<src,dst>, COST)
3:     return cost
4: end procedure
```

**Algorithm 4** Example of available bandwidth cost function

**Ensure:** Integer value representing the result of this Cost Function.

```
1: MAX_BW the highest theoretical available bandwidth in the network
2: procedure AVAILABLE_BW_CF(src, dst)
3:     abw ← get_attribute(<src,dst>, ABW)
4:     return (MAX_BW – abw)
5: end procedure
```

`REACHABILITY` attribute equal to 1 if it is reachable. Otherwise, the attribute is set to the maximum integer value. The cost function `is_reachable_cf`, implemented in Algorithm 1, thus makes reachable destinations more preferable than unreachable ones.

The locality of a path is determined by the originating AS number of the path destination. If the destination prefix is originated by the operator, the path is considered local. Algorithm 2 shows how to implement the `locality_cf` cost function that prefers local paths over non-local ones. In this function, `LOCAL_ASN` is operator AS number.

Algorithm 3 shows the `minimize_cost_cf` cost function that returns the monetary cost of using a path. This function makes path with the lowest monetary cost more attractive. To avoid oscillations, it is a good idea to use classes of monetary costs instead of the exact monetary cost. For example, the `COST` attribute could be the monetary cost modulo $x$ instead of being the raw value of the monetary cost.

When considering bandwidth, the best paths are those having the highest available bandwidth. The implementation of a cost function preferring

**Algorithm 5** Example of customer family cost function

**Ensure:** Integer value representing the customer family for traffic from *src* to *dst*.

```
1: procedure CUSTOMER_FAMILY_CF(src, dst)
2:     family ← get_attribute(<src,dst>, FAMILY)
3:     return family
4: end procedure
```

**Algorithm 6** Example of a complex cost function

**Ensure:** Encounters customers requirements

```
1: PREMIUM_USER = 1
2: STANDARD_USER = 10
3: LIGHT_USER = 20
4: procedure CUSTOMER_MANAGEMENT_CF(src, dst)
5:     if (is_reachable_cf (src, dst) = MAX_INTEGER) then
6:         return (UNREACHABLE)
7:     end if
8:     customer ← CUSTOMER_FAMILY_CF(src, dst)
9:     if (customer = PREMIUM_USER) then
10:        cost ← AVAILABLE_BW_CF(src, dst)
11:    end if
12:    if ((customer = STANDARD_USER ∧ DAY) ∨ customer =
           LIGHT_USER) then
13:        cost ← MINIMIZE_COST_CF(src, dst)
14:    end if
15:    if (customer = STANDARD_USER ∧ NIGHT) then
16:        cost ← AVAILABLE_BW_CF(src, dst)
17:    end if
18:    return ( LOCALITY_CF(src, dst) · cost ) + cost
19: end procedure
```

paths with the highest bandwidth is not straightforward. Indeed, IDIPS, by definition, always prefers the lowest cost while in terms of bandwidth, the highest is the best. Thus, to prefer the paths with the highest bandwidth, the value of the available bandwidth is subtracted to the highest theoretical available bandwidth for the operator (i.e., the capacity of the best link (or link bundle) in the network). Algorithm 4 provides the implementation of such a cost function, `MAX_BW` being the highest theoretical available bandwidth in the network.

As for cost minimization, the customer family cost function only has to return the customer family. Algorithm 5 shows the implementation of this cost function. In the system, the family 1 corresponds to premium users, 10 is for standard users and 20 for light users.

The previous algorithms can be combined by the network operator to build more complex policies. Algorithm 6 combines all the blocks in order to reflect the operator policies proposed earlier in this section. In particular, Algorithm 6 first checks whether the destination `dst` is reachable from the source `src`. If the path is reachable, it applies the policies previously defined, based on the `FAMILY` attribute. For *premium* clients available bandwidth is always preferred. For *standard* clients the applied policy depends on the time period; the available bandwidth is used as cost function during the night, while cost minimization is preferred during the day.

The last line gives preference to a local paths. This line is an example of weighted sum of cost functions. More particularly, the cost result by the `CUSTOMER_MANAGEMENT_CF` is a weighted sum of the costs from other cost functions, weight by the cost returned by a cost function. The principle in the example is to double the cost if the path is not local.

# 4 Performance Based Traffic Engineering with LISP and IDIPS

Sec. **??** shows how the priority, weight and TTL can be used to ensure traffic engineering with LISP. In LISP, the best locator is the locator presenting the lowest priority value. To some extent, we can thus consider the RLOC priority as its relative rank in the mapping. As a consequence, IDIPS ranks can be used to infer the RLOC priorities dynamically. Using IDIPS to attribute the LISP priorities then enables automated performance based incoming traffic engineering. Sec. **??** shows that EID prefixes are likely to be attached to several locators. If an EID have several locators, which are not aliases, it means that it exists several paths to exchange packets with it. However, it is well known that the paths on the Internet are not equal [AAS03, AMS+08, GDZ06, Int05, Cis, SAA+99, Ava05, XYK+08, DD06]. . Some path have lower delay than others, more bandwidth or are more stable. Fig. 15 shows that this observation holds with LISP.

Fig. 15, is obtained from the ping dataset we built in Sec. **??**. Fig. 15 shows the relative difference of delay between all the locators of a mapping and the locator with the shorter delay in the mapping. The relative distance of the locator $i$ with the fastest locator $f$ of the mapping is computed by $\frac{rtt_i - rtt_f}{rtt_f}$. A 10% distance thus means that the locator has a delay higher by 10% than the locator with the shorter delay in the mapping. Fig. 15 shows the average distance to the shorter delay locator in the mapping and the $95^{th}$ confidence interval. Fig. 15 shows that for 4% of the mappings, the locators had a delay more than 50% bigger than the shortest delay observed in the mappings with 1% of the mappings with locators presenting a delay more than twice longer than the shortest one. For few mappings, the average distance was even worst by more than 500% on average up to 6309% higher

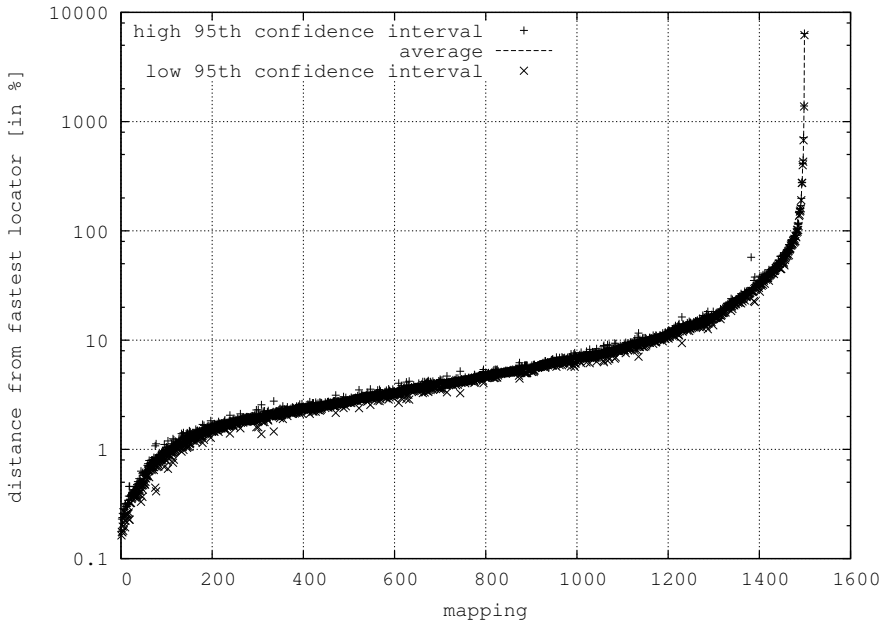



Figure 15: Relative distance between the locators in the mappings

on average. Fig. 15 shows that a bad choice of locator can have a significant impact on the performance showing the necessity of assigning the priorities with care. However, Fig 15 also shows that for 77% of the mappings, the average distance to the locator with the shortest delay in the mapping is lower than 10%. It thus means that for most of the mappings, traffic can be load balanced between at least two locators by playing with the weight without degrading the performances [KKSB07].

Fig. 15 shows that the traffic could benefit from a traffic engineering approach that considers performances as much as costs. IDIPS can help in the choice of the priorities an the weights in LISP. IDIPS is however designed to rank paths while LISP mappings are related to IP addresses. But assuming that the locator priority has to be computed for a mapping to be distributed on the mapping system, the RLOC $R$ to set the priority to can be seen as the destination of a path. If differentiated mapping is not used, this path is then defined as `<*,R>` where `*` corresponds to any source in the Internet. If differentiated mapping is used and the differentiation is done for the site $S$, then, the path is `<S,R>`. As we have seen in Chapter. **??**, IDIPS supports paths defined by prefixes. Therefore, if $P_S$ is the prefix for the site $S$, then the path can be defined as `<`$P_S$`,R/32>`. $P_S$ being 0.0.0.0/0 for `*`.

As a result, IDIPS can be used to set the RLOC priorities but a translation work is required. Two options are possible to determine the priorities

in a mapping. Either IDIPS is queried manually and an operator configures LISP with the appropriate priorities, or a tool automatically adapts the priorities.

Operations related to interdomain traffic engineering are often performed manually by the network operators [CGG+04]. The operators configure their network such that it corresponds to the business objective and react to alarm triggered by monitoring tools. Manual operation can be continued with IDIPS used as a hint by the operator. Before setting the priorities, the operator requests IDIPS to rank the paths corresponding to the locators in the mapping. The operator then sets the priorities to reflect his interpretation of the IDIPS ranks. For instance, he can request IDIPS twice. Once with delay optimization and once with cost minimization. The best result(s) of each query being set with a priority of one, all the others with a priority of 10. Manual IDIPS-aided operation is interesting to support subjective criteria but is subject to configuration error and is time consuming in complex networks [WSR09, CGG+04].

If a translation tool is used between LISP and IDIPS, it is possible to fully automate the IDIPS-based priority computation. The tool listens the insert and update events in the EID-to-RLOC Database. The locators in the mapping that triggered the event are then extracted to build the destination list. The source list depends on whether the differentiated mapping is used or not. Once the source and destination lists are built, the tool requests IDIPS to rank these paths according to a pre-configured ranking criterion. Upon IDIPS reply, the tool translates the IDIPS rank to LISP priority by applying a $t$ translation function on each rank such that:

$$t : \mathcal{R} \times \mathcal{I} \rightarrow \mathcal{P}$$

where $\mathcal{R} = [0; 2^{32}[ \subset \mathbb{R}_0$ and $\mathcal{P} = [0; 255] \subset \mathbb{R}_0$. $\mathcal{R}$ is the IDIPS ranking space and $\mathcal{P}$ is the LISP priority space. $\mathcal{I} \subseteq \mathcal{IP} \times \mathcal{IP} \times \mathcal{R}$ is the space of all the possible IDIPS replies with $\mathcal{IP}$ being the IP space.

$t$ can be any function and is defined according to the network requirements. For example, if only the best locator can be used, i.e., the other cannot, $t$ can be defined as follow:

$$t(x, m) = \begin{cases} 1 & \text{if } x = min(m) \\ 255 & \text{otherwise} \end{cases}$$

where $min(m) : \mathcal{I} \rightarrow \mathcal{R}$ gives the minimum rank observed in the IDIPS reply $m$.

Once the ranks have been converted into priorities, the mapping can be built. To do so, the rank of each locator $r$ in the mapping is extracted from the IDIPS reply and converted into the associated priority. If the locator is not present in the IDIPS reply, it must be considered as non-reachable and should not be present in the mapping or with a priority set to 255.

In addition to the priority, a weight is associated to each locator in a mapping. However, an IDIPS reply only provides priority information. A first solution is to set the same weight to all the locators with the same priority, e.g., $1/n$ when $n$ locators have the same priority. This weight assignment can be used when the rank computed by IDIPS depends on the link usage. Therefore, IDIPS will eventually remove the locators that have been used too much or at least give them an higher rank value. A second solution is to compute the weights independently of IDIPS. A third solution is to define an IDIPS cost function, IDIPS can thus be used to indirectly compute the weight. Let us call this function `proportion_cf`. `proportion_cf` is a cost function which provides a rank corresponding to the proportion of traffic to send between the paths. For example, if the path $p_1$ must support one half of the traffic while $p_2$ and $p_3$ must handle both 25% of the traffic, the ranks could be 2, 1 and 1 respectively. Which means that path $p_1$ must handle two times more traffic than paths $p_1$ or $p_2$. However, the weights in LISP must be such that the sum of the weights of the locators with the same priority must be equal to 100. As a consequence, the ranks from `proportion_cf` must be converted into weights. To do so, IDIPS is first queried with the list of locators to rank for the priority. IDIPS is then queried to rank them with the `proportion_cf` cost function. Afterward, the locators are grouped by priority. Let $R = \{r_1, \dots r_n\}$ the set of locators with the same priority. The weight of locator $r_i \in R$ is defined by

$$weight(r_i) = \left\lfloor \frac{rank(r_i)}{\sum_{r_j \in R} rank(r_j)} \right\rfloor \cdot 100$$

where $rank(x)$ gives the rank of locator $x$ as defined by IDIPS. However, because LISP works with integer weights, the conversion does not ensure that the sum of the weights is 100. For the sum to equal 100, we suggest to use the following smoothing method:

$$weight(r_m) = weight(r_m) + \left( 100 - \sum_{r_j \in R} weight(r_j) \right)$$

where $r_m$ is a locator with the highest weight.

The mapping TTL should be lower or equal to the IDIPS replied TTL. Therefore, when the mapping expires in the EID-to-RLOC database, IDIPS must be queried to re-build the mapping with up-to-date information obtained from IDIPS.

IDIPS can work with prefixes to reduce the number and size of the exchanged messages. If IDIPS replies with prefixes instead of addresses, it becomes possible to use a reply for different EIDs. For instance, if an EID has RLOCs within prefixes already returned by IDIPS, it is possible to reuse the ranking. This technique has two advantages. First, it reduces the traffic

to and from IDIPS and, second, it reduces the time required to obtain the optimal RLOC priorities. A drawback is that a specific cache has to be implemented on the LISP router.

### 4.1 Case Study

In this section, we evaluate the benefits of the interaction between LISP and IDIPS. To do so, we build the testbed depicted in Fig. 16. The left hand network, labeled *Content Producer*, is a content producer and the right hand network is the consumer. Interdomain connectivity is ensured by LISP. For the test, we used the two types of customers *light* and *premium* and apply the `customer_management_cf` cost function presented in Sec. 3.3. As discussed in Sec. 3.3, the objective for light users is cost reduction. On the contrary, QoS has to be ensured for premium users. In the sake of clarity, in our experiments two clients with one flow per client are involved. The light client downloads a large file using FTP (TCP) while the premium client watches a video over UDP. The video must have at least a 1.4Mbps bandwidth and the jitter must be limited. The two networks are connected with two links: L1 and L2. L1 represents a peering link and L2 a customer/provider link (from the producer point of view). L2 is protected by a 128Kbps backup link. Penalties are due when QoS is not ensured for premium users.

In the testbed, we use OPENLISP [ISB11]. OPENLISP implements the LISP protocol in the FreeBSD kernel. A particularity of OPENLISP is the *mapping socket* that allows user space applications to interact with the EID-to-RLOC mappings maintained in the kernel.

The content producer (resp. consumer) part of the testbed is operated with a dedicated Pentium 4 computer running FreeBSD and OpenLISP. The machine is used at the same time as xTR and as content producer (resp. consumer). The two machines are connected via a third machine that runs FreeBSD and dummynet to emulate the links [Riz97].

An IDIPS server instance runs in each network. At that point, neither OPENLISP nor IDIPS are aware of each other. A wrapper runs on each OPENLISP router to allow LISP to query IDIPS. The wrapper monitors the mapping socket and the IDIPS control plane. When there is an event concerning an EID of the local OPENLISP's map table, the wrapper retrieves all the RLOCs for that EID and asks the IDIPS server to rank them. The resulted ranks are translated into priorities and the EID's mapping is updated according to the information given by IDIPS.

The experiment is divided in four periods ($P_1$ to $P_4$). The RLOCs used for each period depends on the IDIPS rankings. During $P_1$, both L1 and L2 are working properly and IDIPS optimizes the performance for premium traffic and minimizes the cost for the light traffic. The beginning of $P_2$ corresponds to the L2 link failure: L2 traffic is diverted to the backup link. During L2, IDIPS is not involved and the RLOCs are not modified, premium

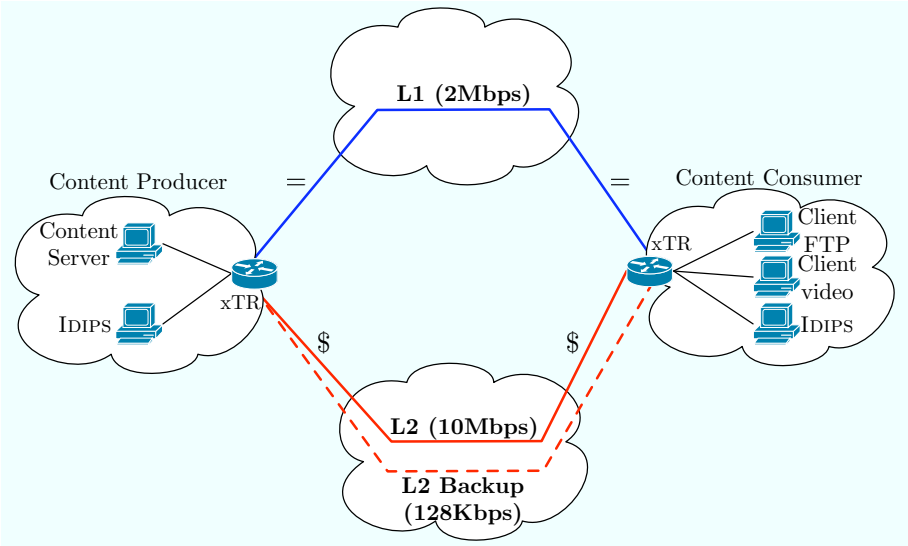



Figure 16: Case study testbed

traffic is degraded. In $P_3$, IDIPS is informed of the failure and modifies the mapping to minimize the cost and avoid backup links. It is worth to notice that the gap between $P_1$ and $P_3$ is for illustration only, in practice, IDIPS can be informed of the topology change at the same time as the backup link activation (e.g., via SNMP). During $P_3$, the backup link is not used anymore. Finally, $P_4$ shows what happens if IDIPS policies are set to the original premium and light traffic requirements (as during $P_1$). In $P_4$, IDIPS decides to divert the light traffic (i.e., FTP) to the backup link and keep premium traffic on L1 to ensure its QoS requirements while minimizing costs. For the experiment, IPFW is instantiated on each machine to monitor the link usage thank to the IPFW statistics [LLB02, Riz97]. The volume monitored by IPFW correspond to the amount of traffic that crossed the links.

Fig. 17 shows the flows' dynamic during the different periods. The horizontal axis is the normalized time and the vertical axis the bandwidth (in Kbps). The best effort traffic consist of a big file transfer using FTP (TCP). The video is simulated with Iperf. Iperf continuously sends 700 bytes long UDP segments with a constant rate of 1.7 Mbps.

During $P_1$, both flows are working as expected: the video (premium customer) encounters a limited jitter and has enough bandwidth (1.7Mbps) and the cost for FTP (light customer) is minimized. After the failure, during $P_2$, the video stream is redirected to the backup link. The video flow bandwidth falls down to around 100Kbps, which is not sufficient to ensure QoS (1.4 Mbps is required to ensure QoS requirements). FTP traffic is not affected by the failure as it is carried by L1. $P_3$ presents the flow bandwidth

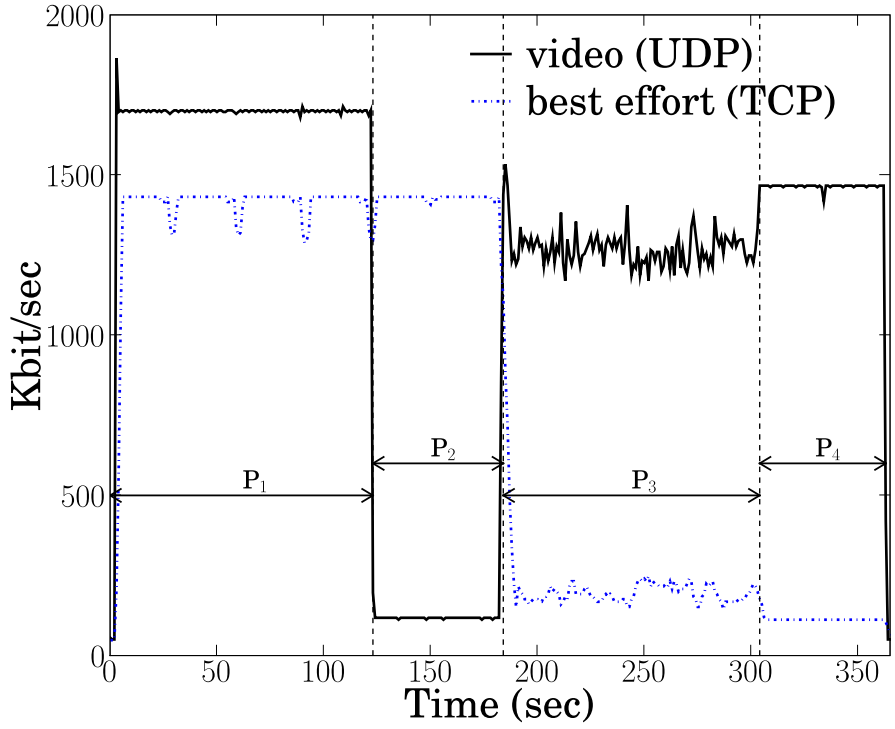



Figure 17: Evolution of the different flows bandwidth for the different network events.

when all the traffic is diverted on L1. For that period, the policies in IDIPS are to avoid backup links. However, this choice does not ensure QoS for the video as the jitter is important and video bandwidth falls to 1.3Mbps. With this configuration, video traffic is influenced by the TCP behavior of the FTP flow. Period $P_4$ shows what happens if IDIPS is configured to ensure QoS and minimize costs, thus video is diverted on link L1 as this is the only one allowing QoS for video. The best effort flow is diverted to L2 backup link because the costs of using it is lower than the cost of losing QoS for video.

This test shows that IDIPS path selection algorithm can take administrative and technical question into account (e.g., minimize costs but maximize bandwidth). Furthermore, it also shows that the simplicity of IDIPS allows to use it in situations where several paths are possible.

# 5 Evaluation

In this section, we evaluate the performance of our IDIPS implementation. We first present the methodology we follow and the testbed we build (Sec. 5.1). The testbed is based on the XORP implementation of IDIPS we

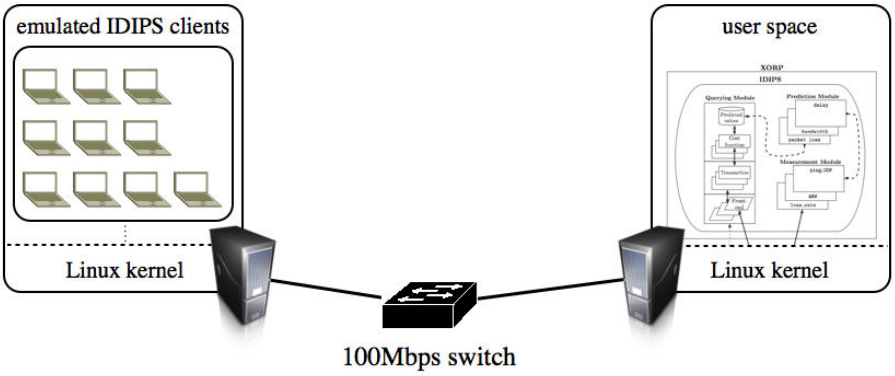



Figure 18: Evaluation testbed

discussed in Sec. 3.2. We next discuss the results (Sec. 5.2). In particular, we focus on the heart of IDIPS, i.e., the ranking process.

## 5.1 Methodology

Fig. 18 provides a description of the testbed we build for evaluating IDIPS. The server on which IDIPS is running is a quad-core Xeon E5430 at 2.66GHz. The server has a 6MB processor cache and has 4GB of RAM. The server runs a Linux 2.6.31-22-server 64 bits distribution. The predicted values storage is populated with 1,500,000 paths randomly generated, each path having a single delay randomly assigned in [0, 1000ms]. The cost function implemented is the `minimize_delay_cf(src, dst)`.

The client is running a dual-core Xeon 3060 at 2.40GHz, with 5GB of RAM and a Linux 2.6.32-5 32 bits distribution. If there is a single hardware representing the client, the machine ensures that 100 instances of the client are running in parallel. Each client instance sends a request to IDIPS, each request containing a source address and a list of destination addresses. Those destinations are randomly generated but we ensure that roughly 10% of the randomly generated destinations are not present in the paths stored by IDIPS. Each instance of the client is in charged of sending 10,000 requests to IDIPS, meaning that a total of 1,000,000 requests are sent to IDIPS. We consider the following values for the number of destinations in a request: 1, 2, 5, 10, 50 and 100.

The clients and the IDIPS server are are attached with a 100Mbps switch.

Each experiment in Sec. 5.2 is repeated ten times. Each data point represents the mean value over ten runs of the experiment, the clients and the server process being rebooted before each run. We determine $95^{\text{th}}$ confidence intervals for the mean based, since the sample size is relatively small, on the Student $t$ distribution. These intervals are typically, though not in all cases,

too tight to appear on the plots.

### 5.2 Results

Fig. 19 shows the IDIPS service time from the client perspective. In particular, Fig. 19(a) gives the time (in ms) between each request and the associated reply for a particular run when the client request contains ten destinations. Fig. 19(a) shows how IDIPS is stable over transactions, no drift is observed.

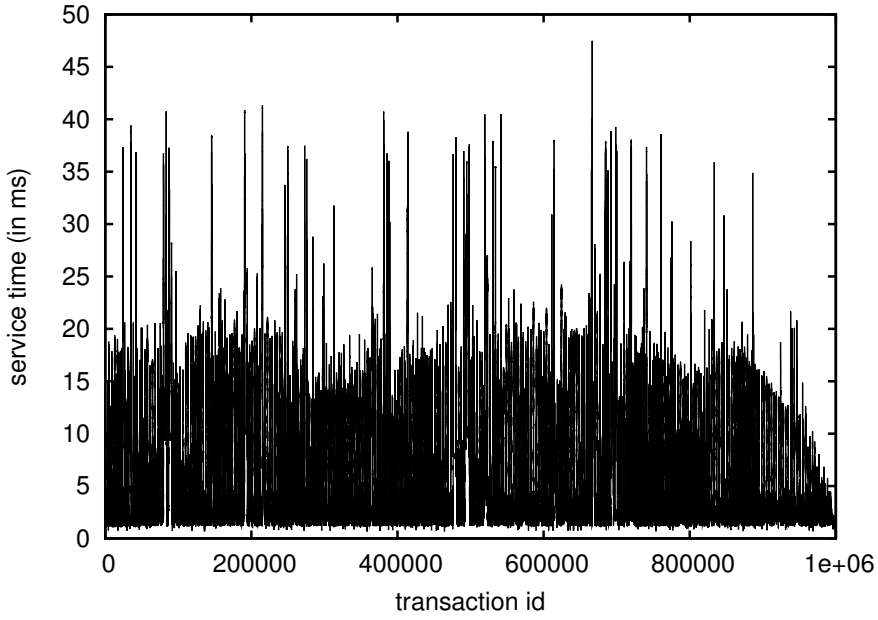
Fig. 19(b) shows the IDIPS time distribution as quantiles. The dotted line represents the median value, while the box plot gives the minimum and $95^{\text{th}}$ percentile values as well as the $25^{\text{th}}$ and $75^{\text{th}}$ percentiles.

Obviously, the IDIPS service time increases with the number of paths (i.e., the number of destinations in this example) induced by the client requests. The service time linearly increases with the number of paths. The linear dependency is the result of the conversion of the list received from the client in text into a binary format into the querying module implementation, the construction of the possible paths and the cost computation for each of such paths. The cost function having a temporal complexity of $\mathcal{O}(1)$, the total complexity is $\mathcal{O}(s * d) = \mathcal{O}(n)$ where $s$ is the number of sources in the request, $d$, the number of destinations and $n$ the number of paths. In our experiment, $s = 1$ making $n = d$. In addition, the bigger the number of destinations, the less stable the service time as suggested by the service time distribution amplitude. The higher dispersion observed for the list of one destination can be explained by the overhead caused by the switching between the XORP processes (i.e., the finder and the querying module).

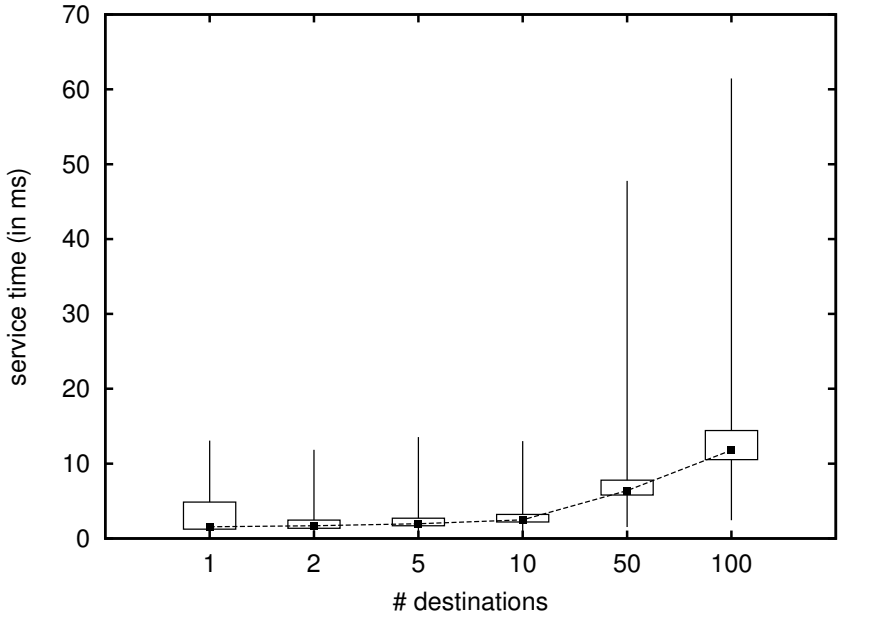
Fig. 20 breaks the IDIPS service time down into three categories: the network delay (labeled "network" - descending line pattern portion of the stacked bars), the path ranking (labeled "IDIPS" - ascending dashed line pattern portion of the stacked bars), and the internal XORP processing (labeled "XORP" - descending dashed line pattern portion of the stacked bars). For plotting those results, we consider the median value among the ten runs. Instead of plotting the median value of the service time, we rather consider the time proportion of each category.

The time consumed by the network is negligible. This is due to clients and server, in our testbed, are separated by a single switch. However, as the IDIPS server is supposed to be deployed within a campus or an ISP network (in the fashion of DNS service), one can imagine that the required network time (i.e., time spend in the network between the client and the server and vice-versa) would be very close to what we experienced in our testbed.

In general, the time spent in the whole ranking process in IDIPS increases linearly with the number of paths from the requests. With about only 12% to 20% of the time spent building the list to return to the client once the costs are computed. The rest being spent by the cost computation and attribute




(a) service time stability - destinations=10




(b) service time distribution

Figure 19: IDIPS service time as perceived by the client

retrieval. The internal XORP processing represents most of the service time

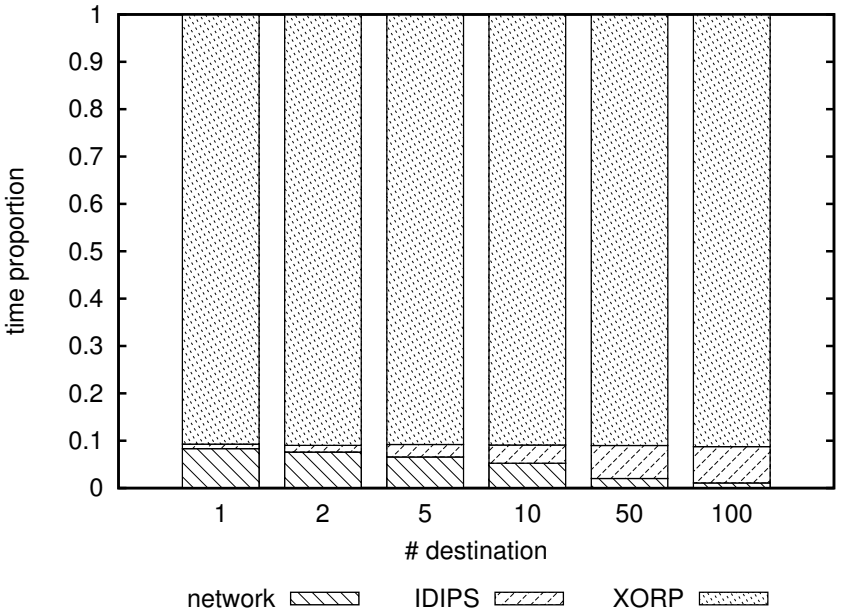



Figure 20: Proportion of the service time split - median over the ten runs

(around 90%) and is also linearly dependent with the number of paths from the requests. The time spent directly in the XORP internals is mostly due because of the marshaling and unmarshaling of the XRLs and the context switching between the `finder` and the querying module (remember that the XRLs are always processed by the `finder`.

Fig. 21 shows the load on IDIPS in term of requests/second number. Obviously, the capacity of IDIPS to process requests decreases with the request size. It is a normal behavior as large requests require more processing time, in terms of IDIPS (typically more cost function to evaluate and, thus, more lookup into the predicted values storage) and internal XORP processing (as already suggested by Fig. 20).

We also notice that, in the worst case (i.e., 100 destinations per request), IDIPS can still process more request per second than what could be required for peer-to-peer applications [GCX+05].

# 6 Related Work

In 2007, when we started to design IDIPS P2P optimization was not an objective [SDIB08]. However, IDIPS can help to select the best peers in P2P systems. Despite the fact that we do not use IDIPS for P2P traffic optimization, most of the IDIPS related work are in that field.

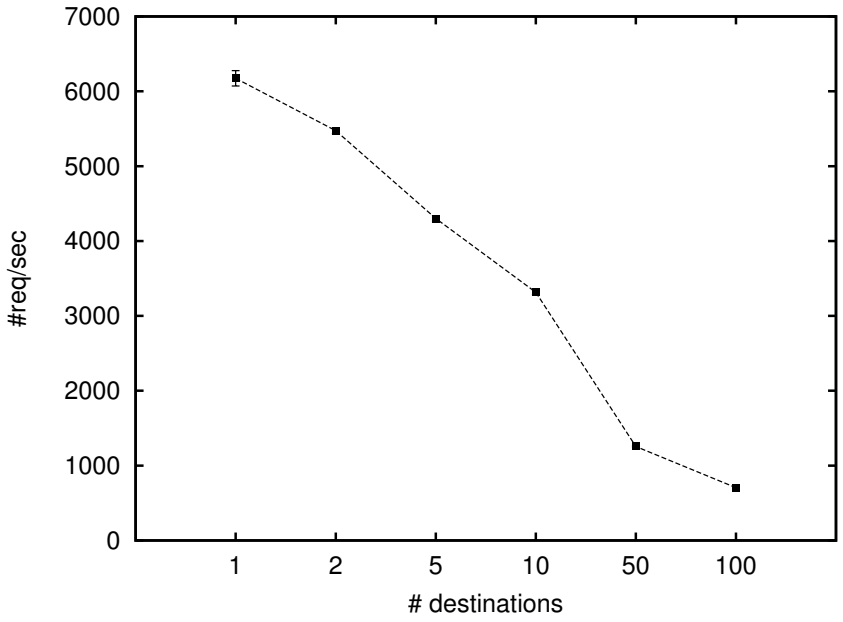



Figure 21: Load on IDIPS without XRL

Bindal et al. show by simulation that biasing the peer choice in P2P can improve the overall performance [BCC+06]. On the one hand, download times can be reduced and on the other hand ISPs can reduce their operational costs. This simulation based study shows that an informed peer selection gives better results than an unbiased peer selection when using Bittorrent [PGES05].

Aggarwal et al. [AFS07] propose an *oracle* service very similar to IDIPS that would be configured by the network operator and queried by P2P applications [AFS07]. The underlying assumption in oracle is that traffic would be optimized by letting ISPs and P2P clients collaborate. The oracle is a server operated by the network operator. When a P2P client needs to select peers in a swarm, it sends the list of peer's IP addresses to the oracle. The oracle returns these addresses labeled with a rank. The way the rank is computed by the oracle is hidden to the clients. The paper shows that preferring peers within the local AS increases the overall performance and should reduce the operational costs for the ISP as the traffic tends to remains local. [AFS07] characterizes performance metrics to evaluate the benefits of using a biased peer selection like oracle. While the oracle limits the ranking to destination addresses, IDIPS proposes to rank paths or even groups of paths, using prefixes. Another difference between the oracle and IDIPS is the ranking scope: the oracle ranking is limited to local or peering domains while such a limitation does not exists in IDIPS.

Choffnes and Bustamante [CB08], as opposed to the oracle approach [AFS07], propose a biased peer selection in P2P without the need of a specific infrastructure. Instead, Choffnes and Bustamante propose *Ono* that uses the information that are already available within commercial CDNs. In general, CDNs like Akamai [5] and Limelight [6] control the traffic from the clients by using the DNS. To be simple, depending on the client location, the DNS replies for a name are different. By periodically resolving names for CDNs servers, the peers can build an abstracted representation of the relations they have with these CDNs, this representation is called the *behavior map*. Each Ono-enabled peer resolves the name of the same 6 well chosen CDN names. The corresponding C-class prefix is considered instead of the addresses. Indeed CDNs often have several servers in the same data-center and the servers are grouped in C-class prefixes. The addresses belonging to the same C-class prefix can be safely considered as belonging to the same datacenter. The peers do the resolution several times to determine the dynamic of the change in the CDN name map. Two peers "close" to one name will have the same dynamics in the CDN map. In addition, to the change ratio of the CDN names in the map, the peers ping the returned addresses. The measured delay is used to weight the map according to the delay with the servers. When a peer must select another peer, it will chose a peer that presents a similar weight ratio map. The RTT weighting relies on the principle that the closer two peers are, the more likely the name will be resolved in the same /24 subnet. and the delay to the server for a given name will be close as well. The deployment of Ono on hundred of thousands nodes shows that doing so can reduce the average delay and increase the download rate by about 30%. It also shows that in 33% of the cases, the chosen peers were within the local ISP. The major difference between IDIPS and Ono is that no specific infrastructure must be deployed for the service to work. However, the reduction of inter-AS cost is only a side effect of the optimization of the perceived client performance. Ono does not leverage an ISP P2P collaboration.

Xie et al. propose *P4P* [XYK+08]. P4P provides topology hints to P2P clients. In P4P, the topology and policies are abstracted into the p-distance. The p-distance summarizes the metrics that are relevant for the operator and is used by ISPs to communicate their preferences and status for the traffic. The application uses the p-distance to create an abstracted view of the network connectivity. The applications can then select their destinations based on the minimal p-distance in the abstracted network view. ISPs, or third parties, maintain iTrackers to communicate the p-values. These iTrackers are contacted by P2P clients to compute the set of the best swarm peers. The iTrackers are used to retrieve the policy information, the p-values

[5] `http://www.akamai.com`

[6] `http://www.limelightnetworks.com`

but also the network capabilities.

During the mid 2000ies, network operator started to complain about the operational cost of supporting P2P traffic [Lig, She, KTCI04]. Indeed, the P2P overlays do not take the underlay network topology into account and neighbors are in general selected at random. Therefore, it is possible that a packet sent by a peer in an AS A to a peer in the same AS follows a path that crosses several times the network boundaries, possibly via provider links. Providers have then started to block P2P traffic or to limit the bandwidth for P2P flows. However, P2P application now rely on encryption or simply behave like HTTP to masquerade their traffic. The problems faced by the ISPs is because ISPs are black boxes and it is hard for a client application to know the peers that are efficient but not harmful for its ISP. Industry and academy started to collaborate on a solution for this problem with an IETF P2P Workshop meeting specially hold in May 2008 in Boston.[7] During this workshop researchers and operators were invited to present position papers about the problem and the solutions they though to be good for this. The conclusion of the workshop is that P2P and ISPs must collaborate somehow. Some proposed to add caches and install the trackers inside the ASes, but this would have caused legal issues as it is known that P2P traffic is mostly used for illegal content. As a result, the *Application-Layer Traffic Optimization* (ALTO) [SB09] Working Group as been set up at the IETF. ALTO aims at designing and specifying services helping applications such as peer-to-peer, content delivery networks, and mirror selection, to select the best peers. Factors of interest in this selection are, among others, maximum bandwidth, minimum cross-domain traffic, lowest cost to the user, etc. ALTO is a mix of the best features of P4P, oracle and IDIPS with additional contributions. Clearly, IDIPS is in line with the ALTO Working Group thematic. However, while we have been actively working in the ALTO working group at its beginning [AFP+09] to make sure that the protocol would be able to support IP prefixes, names or AS numbers, we have chosen not to implement the ALTO protocol in IDIPS because many features are P2P related and because ALTO is closer to P4P than to oracle or IDIPS. Nevertheless, IDIPS could be adapted to implement the ALTO service as it meets most of the ALTO requirements [KPS+11].

As opposed to oracle, Ono, P4P and ALTO in general, IDIPS is not designed specifically for P2P. IDIPS is an extension of the *NAROS Name, Address and ROute System (NAROS)* server that has been designed for multihoming IPv6 host-centric traffic engineering in the early 2000 [dLBL03]. NAROS was proposed to select the best source address in multihomed IPv6 sites. A NAROS client sends a request to a NAROS server with the destination address it aims to use and the list of its source addresses. The server returns the best source address to use for the destination. The pre-

[7] `http://www.funchords.com/p2pi/`

fix of the destination is also returned to avoid a client requesting several times the server for destinations within the same prefix and that would thus have the same result. The major difference between IDIPS and NAROS is that IDIPS return all the possible aggregated <source,destination> pairs and ranks them. This change allows IDIPS to be used equally for incoming and outgoing traffic engineering and enables the use of multi-path routing or transport.

A proposal that shares objectives similar to IDIPS is *Morpheus* [WAR07], which determines the best path to use according to the operator policies and, then, sends BGP updates to its BGP router target (via multihop eBGP). Like IDIPS, Morpheus is very modular but is restricted to BGP as the signaling is performed using BGP messages while IDIPS has its own messaging format, allowing a finer-grained interaction between the client and the path selection service.

Finally, a number of vendors have proposed proprietary path selection solution ([Int05, Ava05, Rad, Cis]). These solutions all follow the same principle. Specialized boxes are deployed in the network and monitor it actively, passively or both. The measurements are combined with policies to determine the quality of the different routes. Based on the observations and the configuration, the boxes can inject prefixes into BGP to influence the incoming traffic but they can also modify link costs in the IGP or inject partial BGP tables into it to control the outgoing traffic as well. Finally, NAT can be used to ensure that some flows enter the network via a given link.

# 7 Conclusion

The Internet is evolving. During this last decade, we have seen the emergence of applications having more and more requirements in terms of delay, jitter,or bandwidth. In addition, the single path assumption between a source and a destination does not hold anymore. As a consequence, the applications can use several paths to retrieve their content. It might thus be interesting to provide those applications a service for selecting their paths better than randomly, i.e., a service for selecting paths that meet applications requirements.

In this chapter, we proposed an Informed Path Selection Service (IDIPS), that is able to rank paths. IDIPS is a generic, scalable, lightweight and easily deployable solution allowing ISPs, enterprises, or campus networks to qualify paths between a source and a set of destinations. IDIPS makes use of passive and active measurements to keep track of the network conditions.

In this chapter, we have shown that IDIPS can be used to enable performance based incoming traffic engineering with LISP.

We discussed our IDIPS implementation inside XORP and focused on

simple cost function (i.e., the way IDIPS assigns a cost to a path) construction and how to combine them to reflect more complex ranking strategies. We built a testbed and evaluated the performance of IDIPS. In particular, we focused on the heart of IDIPS, the ranking process. We demonstrated that IDIPS is robust as it is able to process a large number of requests/second while providing a stable response time to the client.

## Acknowledgments

This work is supported by the ECODE European Project

## References

[AAS03] A. Akella, Shaikh A., and R. Sitaraman. A measurement-based analysis of multihoming. In *Proc. ACM SIGCOMM*, August 2003.

[ACK03] S. Agarwal, C-N. Chuah, and R. H. Katz. OPCA: Robust interdomain policy routing and traffic control. In *Proc. IEEE Conference on Open Architecture and Network Programming (OPENARCH)*, April 2003.

[AFP+09] O. Akonjang, A. Feldmann, S. Previdi, B. Davie, and D. Saucez. The PROXIDOR Service. Internet draft, draft-akonjang-alto-proxidor-00, work in progress, March 2009.

[AFS07] V. Aggarwal, A. Feldmann, and C. Scheideler. Can ISPs and P2P users cooperate for improved performance. *ACM SIGCOMM CCR*, 37(3):29–40, July 2007.

[Aka] Akamai. Web application acceleration and performance management, streaming media services, and content delivery. See `http://www.akamai.com`.

[AKZ99] G. Almes, S. Kalidindi, and M. Zekauskas. A Round-trip Delay Metric for IPPM. RFC 2681 (Proposed Standard), September 1999.

[AMS+08] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. On the performance benefits of multihoming route control. *IEEE Transactions on Networking*, 16(1):96–104, February 2008.

[APS04] A. Akella, J. Pang, and A. Shaikh. A Comparison of Overlay Routing and Multihoming Route Control. In *Proceedings of ACM SIGCOMM*, Portland, Oregon, August 2004.

[Ava05] Avaya. Adaptative networking software (ANS), 2005.

[BCC+06] Ruchir Bindal, Pei Cao, William Chan, Jan Medved, George Suwala, Tony Bates, and Amy Zhang. Improving traffic locality in bittorrent via biased neighbor selection. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, pages 66–, Washington, DC, USA, 2006. IEEE Computer Society.

[CB08] David R. Choffnes and Fabián E. Bustamante. Taming the torrent: a practical approach to reducing cross-isp traffic in peer-to-peer systems. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 363–374, New York, NY, USA, 2008. ACM.

[CCRK04] M. Costa, M. Castro, R. Rowstron, and P. Key. PIC: Practical Internet coordinates for distance estimation. In *Proc. 24th International Conference on Distributed Computing Systems*, March 2004.

[CGG+04] Don Caldwell, Anna Gilbert, Joel Gottlieb, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. The cutting edge of ip router configuration. *SIGCOMM Comput. Commun. Rev.*, 34:21–26, January 2004.

[CH10] X. Cai and J. Heidemann. Understanding block-level address usage in the visible Internet. In *Proc. ACM SIGCOMM*, August 2010.

[Cis] Cisco Systems. Optimized edge routing (EOR).

[Cla04] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[CLH03] K. Cho, M. Luckie, and B. Huffaker. Identifying IPv6 network problems in the dual-stack world. In *Proc. ACM SIGCOMM Workshop on Network Troubleshooting*, September 2003.

[DCKM04] F. Dabek, R. Cox, K? Kaashoek, and R. Morris. Vivaldi, a decentralized network coordinated system. In *Proc. ACM SIGCOMM*, August 2004.

[DD06] Amogh Dhamdhere and Constantinos Dovrolis. Isp and egress path selection for multihomed networks. In *IEEE INFOCOM*, 2006.

[DHKS09] Xenofontas Dimitropoulos, Paul Hurley, Andreas Kind, and Marc Stoecklin. On the 95-percentile billing method. In *Passive and Active Measurements Conference (PAM)*, April 2009.

[dLBL03] C. de Launois, O. Bonaventure, and M. Lobelle. The NAROS Approach for IPv6 Multi-homing with Traffic Engineering. In *Proceedings of QoFIS, LNCS 2811, Springer-Verlag*, pages 112–121, October 2003.

[dLUB05] C. de Launois, S. Uhlig, and O. Bonaventure. Scalable route selection for IPv6 multihomed sites. In *Proc. IFIP Networking*, May 2005.

[Dra03] R. Draves. Default address selection for Internet protocol version 6 (IPv6). RFC 3484, Internet Engineering Task Force, February 2003.

[FFM03] M. J. Freedman, E. Freudenthal, and D. Maziéres. Democratizing content pulication with coral. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2003.

[FJP+99] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gruniewicz, and Y. Jin. An architecture for a global Internet host distance estimator service. In *Proc. IEEE INFOCOM*, March 1999.

[GCX+05] Lei Guo, Songqing Chen, Zhen Xiao, Enhua Tan, Xiaoning Ding, and Xiaodong Zhang. Measurements, analysis, and modeling of bittorrent-like systems. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, IMC '05, pages 4–4, Berkeley, CA, USA, 2005. USENIX Association.

[GDZ06] R. Gao, C. Dovrolis, and E. Zegura. Avoiding oscillations due to intelligent route control systems. In *Proc. IEEE INFOCOM*, April 2006.

[HHK03] Mark Handley, Orion Hodson, and Eddie Kohler. Xorp: an open platform for network research. *SIGCOMM Comput. Commun. Rev.*, 33:53–57, January 2003.

[Int05] Internap. Premise-base route optimisation, 2005.

[ISB11] Luigi Iannone, Damien Saucez, and Olivier Bonaventure. Implementing the locator/id separation protocol: Design and experience. *Computer Networks*, 2011. To appear, http://dx.doi.org/10.1016/j.comnet.2010.12.017.

[KKSB07] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37:51–62, March 2007.

[KKY03] R. Katz, K. Kompella, and D. Yeung. Traffic Engineering (TE) Extensions to OSPF Version 2. Internet Engineering Task Force, RFC3630, September 2003.

[KPS+11] S. Kiesel, S. Previdi, M. Stiemerling, R. Woundy, and Y r. Yang. Application-Layer Traffic Optimization (ALTO) Requirements. Internet Draft (Work in Progress) draft-ietf-alto-reqs-08, Internet Engineering Task Force, March 2011.

[KRP05] T. Karagiannis, P. Rodriguez, and K. Papagiannaki. Should Internet service providers fear peer-assisted content distribution. In *Proc. Internet Measurement Conference (IMC)*, October 2005.

[KTCI04] R Keralapura, N Taft, C N Chuah, and G Iannaconne. Can isps take the heat from overlay networks? In *Proceedings of the 3rd Workshop on Hot Topics in Networks (HotNets-III)*, San Diego,, 2004.

[LGP+05] E. K. Lua, T. Griffin, M. Pias, H. Zheng, and J. Crowcroft. On the accuracy of embeddings for Internet coordinate systems. In *Proc. USENIX Internet Measurement Conference (IMC)*, October 2005.

[LGS07] J. Ledlie, P. Gardner, and M. I. Seltzer. Network coordinates in the wild. In *Proc. USENIX Symposium on Networked System Design and Implementation (NSDI)*, April 2007.

[LHC03] H. Lim, J. C. Hou, and C.-H. Choi. Constructing internet coordinate system based on delay measurement. In *Proc. ACM SIGCOMM Internet Measurement Conference (IMC)*, October 2003.

[LHC05] H. Lim, J. C. Hou, and C-H. Choi. Constructing Internet coordinate system based on delay measurement. *IEEE/ACM Transactions on Networking*, 13(3):513–525, June 2005.

[Lig] Light Reading. Controlling P2P Traffic. `http://www.lightreading.com/document.asp?site=lightreading&doc_id=44435&page_number=3`.

[Lim] Limelight Networks. High performances content delivery network for digital media. See `http://www.limelightnetworks.com/`.

[LLB02] Kurt J. Lidl, Deborah G. Lidl, and Paul R. Borman. Flexible packet filtering: providing a rich toolbox. In *Proceedings of the*

*BSD Conference 2002 on BSD Conference*, BSDC'02, pages 11–11, Berkeley, CA, USA, 2002. USENIX Association.

[LPS06] J. Ledlie, P. Pietzuch, and M. I. Seltzer. Stable and accurate network coordinates. In *Proc. International Conference on Distributed Computing Systems*, July 2006.

[MFHK08] A. Matsumoto, T. Fujisaki, R. Hiromi, and K. Kanayama. Problem Statement of Default Address Selection in Multi-prefix Environment: Operational Issues of RFC3484 Default Rules. Internet Draft (Work in Progress) draft-ietf-v6ops-addr-select-ps-05, Internet Engineering Task Force, April 2008.

[MS04] Y. Mao and L. Saul. Modeling distances in large-scale networks by matrix factorization. In *Proc. ACM SIGCOMM Internet Measurement Conference (IMC)*, October 2004.

[NB09] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Proposed Standard), June 2009.

[NZ02] T. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proc. IEEE INFOCOM*, June 2002.

[NZ04] T. S. E. Ng and H. Zhang. A network positioning system for the Internet. In *Proc. USENIX Annual Technical Conference*, June 2004.

[Pap07] V. Pappas. Coordinate-based routing for overlay networks. In *Proc. International Conference on Computer Communications and Networks (ICCCN)*, August 2007.

[PCW+03] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Lighthouses for scalable distributed location. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2003.

[PGES05] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The Bittorrent P2P File-Sharing System: Measurements and Analysis. In Miguel Castro and Robbert van Renesse, editors, *Peer-to-Peer Systems IV*, volume 3640 of *Lecture Notes in Computer Science*, chapter 19, pages 205–216. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[PLMS06] P. Pietzuch, J. Ledlie, M. Mitzenmacher, and M. Seltzer. Network-aware overlays with network coordinates. In *Proc.*

*IEEE International Conference on Distributed Computed Systems Workshops (ICDCSW)*, July 2006.

[Rad] Radware. `http://www.radware.com`.

[Riz97] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):37–41, January 1997.

[RMK+08] V. Ramasubramanian, D. Malhki, F. Kuhn, I. Abraham, M. Balakrishnan, A. Gupta, and A. Akella. A unified network coordinate system for bandwidth and latency. Technical Report MSR-TR-2008-124, Microsoft Research, September 2008.

[SAA+99] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zhorjan. Detour: Informed internet routing and transport. *IEEE Micro*, 19(1):50–59, January/February 1999.

[SB09] J. Seedorf and E. Burger. Application-Layer Traffic Optimization (ALTO) Problem Statement. RFC 5693 (Informational), October 2009.

[SDB09] Damien Saucez, Benoit Donnet, and Olivier Bonaventure. On the impact of clustering on measurement reduction. In *Networking*, pages 835–846, 2009.

[SDIB08] D. Saucez, B. Donnet, L. Iannone, and O. Bonaventure. Interdomain traffic engineering in a locator/identifier separation context. In *Proc. IEEE Internet Network Management Workshop (INM)*, October 2008.

[She] Shen. HPTP: Relieving the Tension between ISPs and P2P. *IPTPS'07*.

[ST03] Y. Shavitt and T. Tankel. Big-bang simulation for embedding network distances in euclidean space. In *Proc. IEEE INFOCOM*, March 2003.

[WAR07] Y. Wang, I. Avramopoulos, and J. Rexford. Morpheus: Making routing programmable. In *Proc. ACM SIGCOMM Workshop on Internet Network Management (INM)*, August 2007.

[WSR09] Yi Wang, Michael Schapira, and Jennifer Rexford. Neighbor-specific bgp: more flexible routing policies while improving global stability. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 217–228, New York, NY, USA, 2009. ACM.

[WSS05] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: a lightweight network location service without virtual coordinates. In *Proc. ACM SIGCOMM*, August 2005.

[XYK+08] H. Xie, Y. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz. P4P: Provider portal for applications. In *Proc. ACM SIGCOMM*, Agust 2008.

[YRCR04] Ming Yang, X. Rong, Li Huimin Chen, and Nageswara S. V. Rao. Predicting internet end-to-end delay: an overview. In *in Proc. of 36th IEEE Southeastern Symposium on Systems Theory*, pages 210–214, 2004.

[ZJUVM07] X. Zhou, M. Jacobsson, H. Uijterwaal, and P. Van Mieghem. IPv6 delay and loss performance evolution. *International Journal of Communication Systems*, 21(6), June 2007.