Implementation and Preliminary Evaluation of IDIPS: technical notes

Damien Saucez, Benoit Donnet, Olivier Bonaventure

October 1, 2007

1 Introduction

The idea behind IDIPS is to provide ordered paths according to pre-defined criteria. IDIPS is composed of three entities: the *client*, the *destination* and the *IDIPS server* itself. The client is the entity sending request for QoS and path information towards a destination. The IDIPS server aims at providing the requested information to the client. IDIPS is operated by an ISP or an AS.

A client sends a list of all the possible source and destination addresses to the IDIPS server. In addition, the client gives its criteria for evaluating paths. When an IDIPS server receives such a request, it creates a set of conceivable $(\text{source}, \text{destination})$ pairs, where each source and destination belongs to the initial list provided by the client. The IDIPS server then orders the pairs according to the client's criteria so that the first pair in the list is the best choice while the last one is the least preferred. The IDIPS server responds to the client by sending back this ordered list. Note that the addresses list provided by the client might be composed of prefixes, not necessarily of complete addresses.

The IDIPS protocol is defined in [1].

This technical paper aims at evaluating our current implementation of the IDIPS server. First, Sec. 2.1 presents the main concepts of the implementation. After, Sec. 2.2 gives some details about the implementation choices. Next, Sec. 3.2 describes the performances testbed and Sec. 3.2 comments the results for this testbed. Finally, Sec. 4 concludes and presents the future implementation's milestones.

2 Implementation

In this section we present our current implementation of the IDIPS server. First, we present the basic concepts used in the implementation. After, we present some technical choices and motivate them.

2.1 Concepts

Our implementation is based on two main concepts: the *knowledge base* (KB) and the *cost function* (CF).

The knowledge base can be seen as a database collecting information about the prefixes. When the server needs information about a prefix, it makes a looks in the KB and immediately obtains the results. No measurement or analysis must be done at that time. The KB must be implemented with a technique that offers efficient look-up for prefixes. Indeed, when processing an `IDIPS_REQUEST`, the lookup time for prefixes must be reduced at its minimum to minimize server's response-time from the client point-of-view. Every prefix in the KB as some attributes associated to it. The attributes are normalized numerical values that represents a specific metric. Normalized metric values follows the same principle as Local Prefs in BGP. The complexity of the metric is hidden behind a numerical value. This numerical value must summary a set of possible various information such that the classification of the prefix for this cost function only consists in sorting prefixes by numerical value of the cost. Because the value of the metrics can vary all the time, a dedicated process is charged to maintain the knowledge base when the information about a prefix is modified. A solution which estimates the metric on client's demand would not be effective as it would increase the response-time of the server. For example, if the metric is the AS_PATH length, an aspahtlength attribute with the normalized length of the AS_PATH is added to prefixes. This is an obvious example, but we could imagine more complex metric.

The cost functions are divided in two pieces. First, a routine is added in the knowledge base maintenance process. This routine is called to recompute the numerical value associated with the metric in the KB. Second, a fast routine is added in the decision-process. This part of the cost function gets a source and a destination prefix as input and returns the cost of the couple. The cost of the couple is a special combination of the metric associated with the source and the same metric associated with the destination. The combination of the two value must be has simple as possible (e.g., a sum) to reduce the response-time of the server. The complexity of the cost function is in the first routine.

When the server receives an IDIPS_REQUEST message, it extracts the source prefixes proposed by the client and the destination prefixes and create all the possible couples. It then computes the cost for each couple and construct an ordered list of couples. The list is ordered by cost. The couple with the lowest cost is the more attractive.

To combine multiple cost functions, a weight is associated to each CF. The global cost of a pair is the weighted sum of its costs. The pair with the lowest cost is the most attractive. When a cost function is not applicable to at least one function, this cost function must be removed from every prefix in the selection process. For example, if the cost function CF2 is not applicable to couple B but applicable to couples A and C. If the server must return the 3 couples A, B and C. The cost function CF2 cannot be used to order the couples.

2.2 Technical choices

The KB is a Patricia tree of IP prefixes in our implementation. Every entry in the KB contains the AS_PATH length and the Local Pref of the prefix.

To update information about prefixes, a process listen a UNIX message queue. The messages sent in the queue contains the prefix, the AS_PATH length and the localpref. When the process receives a message, it update the information for the prefix in the Patricia tree (the knowledge base). If the prefix is not in the KB, it is added with these value as metric.

3 Evaluation

In Sec. 3.1 we present the testbed we used for our experiments. In Sec. 3.2, we depict the results of our experiments and gives some explanation of the results.

3.1 Testbed

Our testbed is composed of two computers connected through a switched fast Ethernet network. The IDIPS server runs on a FreeBSD 5.5 Pentium 4 2.60GHz with 1GB of memory. The clients pool runs on a Linux 2.6.18 Pentium 4 2.80GHz with 1GB of memory.

The BGP information are transmitted to the IDIPS server with an application build over the XORP BGP routes client in the XORP BGP Tools suite. This application is modified to write a message in the message queue of the IDIPS server with the prefix and both its AS_PATH length and its Local Pref.

The experiments are performed for RIBs of different size. The RIBs are those available on Route Views Project web site [2]. We used the following RIBs:
rib.20011026.1648, rib.20030906.0934, rib.20050906.1031 and rib.20070906.1025. This choice of RIBs permits to test the service with a number of routes from 107600 in the 2001's RIB to 244567 in the 2007's RIB. Passing trough 140402 routes in 2003 and 183579 routes in 2005. Route Views does not propose RIBs before October 2001.

Route Views RIBs are dumped into the XORP router with SBGP [3]. The experiments was performed only after the RIBs was completely dumped into IDIPS server. The KB is constant (i.e., entries connot be modified) during an experiment.

We consider a set of 100 clients contacting the IDIPS server. The 100 clients ran on the same computer (e.g., 100 independant processes). Each client periodically generates a list of pseudo-uniform random source and destination prefixes and requests the server for a classification [1]. The interval time between two requests of a particular client is given by a Poisson distribution with a mean

[1] This distribution of prefixes does not reflect the Internet but offers the advantage of browsing KB's entries uniformly.

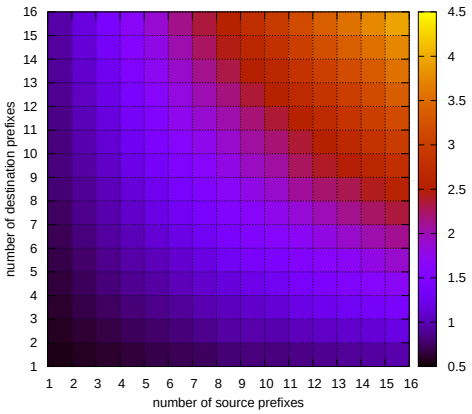

Figure 1: Average processing time as a function of the number of source addresses and the number of destination addresses in the `IDIPS_REQUEST`

of one second. The number of source and destination addresses in the requests follows a uniform random distribution of minimum 0 and maximum 16. Every client sends 50000 requests for each experiment (1 experiment for each RIB). An experiment took approximatively 14 hours.

For every experiment, we stored the following three information: (*i*) the time for the server to process the query without buffer time (the time to find the prefix in the tree and to combine metrics). (*ii*) the time spend between the moment where the client sends the IDIPS request and where it receives the response. (*iii*) the round-trip time from the client pool to the server.

3.2 Results

Fig. 1 shows the evolution of the processing time (color density) in ms as a function of the number of prefixes in the source list (x axis) and the number of destination prefixes in the destination list (y axis). This figure shows that the processing time is symetric when compared with the number of prefixes in source or destination list. For example, the process time for 3 source and 6 destination prefixes is the same as for 6 source and 3 destination prefixes. Until otherwise, in the rest of this document we use the term number of destination prefixes for both source and destination as we only focus on processing time.

Fig. 2 shows the evolution of the processing time (color density) compared to the number of destination prefixes in the request (x axis) and the number of couples of prefixes returned by the server (y axis). This figure point out that the processing time is worse while threating many destination prefixes than few

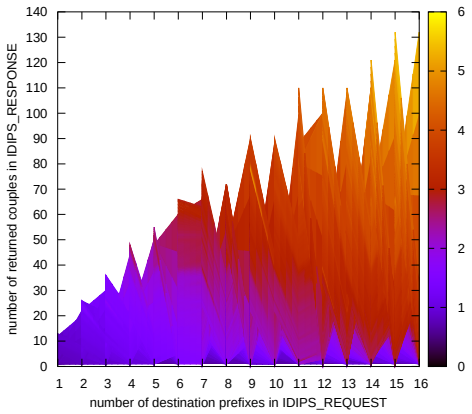

Figure 2: Distribution of the processing time as a function of destination prefixes in the `IDIPS_REQUEST` and returned couples in the `IDIPS_RESPONSE`.

destination prefixes even if the number of returned couples is the same. This can be explain by the implementation which look at every destination prefixes in the list whatever the number of couples are returned. The choice of the maximum number of destination prefixes in the list and the number of couples returned by the server can be based on the precessing time. For example, if the service must have a maximum response-time of 3 ms, the number of destination prefixes in the list must be 6 or 7. Otherwise, the response time could be more than 3 ms. Some peaks are present in the figure because of the invalidity of some couples for the experiments.

This behaviour reflects the implementation that follows the next structure:

```
// process
foreach prefixe in sources
    foreach prefixe in destinations
       get prefixes information
       process couple
    end
end
```

This implementation is sub-optimal as it does not pre-process the prefixes information. A better solution would to retrieve information about the prefixes before the couple creation and, if possible, avoid processing bad couples as presented in the following algorithm

```
// pre-process
```

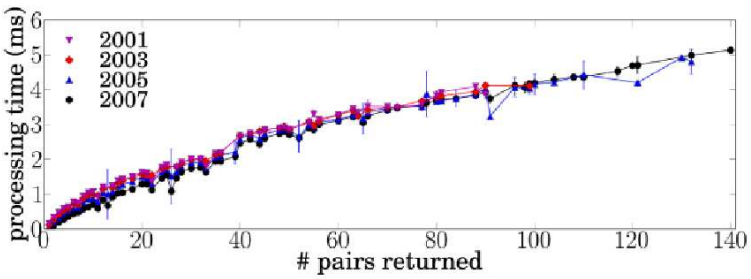

Figure 3: Evolution of the processing time with the number of returned pairs

```
foreach prefixe in sources
    get prefixe information
end
foreach prefixe in destinations
       get prefixe information
end
// process
foreach valid prefixe in sources
    foreach valid prefixe in destinations
       process couple based on pre-process
    end
end
```

Unfortunately, it is hard to determine the validity of a prefixe without creating the couple where the prefixe will be used, i.e., a prefixe could be valid only in conjunction with a specific other prefix while creating a couple. In the next future, a solution must be found to this problem.

For the 2001's RIB, 31% of the requests was rejected (for a total of 5M), 28% for 2003, 23% for 2005 and 17% for 2007. This means that for between 17% and 28% of the request where invalid. A request is considered as invalid if no couple can be formed with the prefixes proposed in the source and destination lists.

We evaluate both the server processing time (the time required by the server to extract the source prefixes and the destination prefixes and create the ordered list of valid couples) and the response-time from the client's point of view. Because the service can run on very different type of networks, the round-trip time is not taken into account for this response-time. For the 2007's RIB, the average processing time at the server side was 716.44 μs. The average response-time at the client-side was 754.3 μs with a round-trip time of 710μs. On average, the end-to-end buffer time (i.e., sum of the time spent in both client and server buffers) was is 37.86 μs.

Fig. 3.2 shows the evolution of the server processing time (in ms, vertical axis) with the number of returned pairs (horizontal axis). The possible pairs are ordered acording to the AS_PATH length metric. Neither clients nor server

maintain caches and the time in buffers is considered as a part of the processing time.

From Fig. 3.2, we see that the IDIPS server processing time oscillates between 0.1 ms (when a single pair is returned) and 4.16 ms (when 100 pairs are returned). We further observe that the size of the returned list never reaches 256 (maximum is 140 for the 2007 curve), i.e., the maximum that can be returned in our tests (all the pairs formed by the 16 sources and 16 destinations). This behaviour comes from the fact that IDIPS server can remove invalid pairs from the returned list. In our experiements, the pairs removed were pairs in which one or the two prefixes (addresses) in the pair have no entry in the knowledge base, i.e., no path exists in the BGP RIB to reach the prefixes. In our tests, we obviously notice that invalid paths are more frequent for small RIBs (e.g., 2001 RIB) than for bigger RIBs (e.g., 2005 RIB).

It is worth to notice that a list of 100 couples has no real interest. Indeed, if the client must use the 100^{th} entry in the list, it means that the previous 99 couples are invalid, which should never occurs. We would recommend to limit the number of returned pairs to 16. In such a situation, the processing time is 1.28 ms on average. Therefore, from a client point of view, the cost associated to IDIPS is negligible, i.e., it is no more expensive than a DNS request.

4 Conclusion

In this technical note, we have briefly presented the IDIPS service, a service that proposes to order paths according to a quality of service metric at a low cost for both the servers and the clients. After, we have describe the main points of our current server's implementation. Basically, the server is devided into two entities. The first collects information about prefixes and convert them into an effective internal representation. The second combines these information to determine the quality of a given path. Based on the results of the combination, different paths can be ordered by quality.

The protocol is defined such that clients give a list of possible source prefixes and a possible list of destination prefixes. The server then constructs the possible couples and computes their quality. Finaly, the server sends back a list of ordered couples where the first entry is the most attractive one and the last is the worst.

Our testbed shows that the quantity of information stored by the server (e.g., the RIB) does not dramatically influence the time-performances. On the contraty, the size of both the lists sent by the clients have impacts on the performances.

This note shows that the implementation can be improved to reduce the response time of the server. In addition, a limit must be set to the size of the lists sent by clients and by servers.

We are currently working in adding effective active measurements metrics to our implementation. In addition, we are looking at a more effective implementation of the cost function aggregator.

References

[1] D. Saucez, B. Donnet, and O. Bonaventure, "IDIPS protocol," Université catholique de Louvain," Draft, September 2007, see http://inl.info.ucl.ac.be/projects/naros.

[2] University of Oregon, "Route views, University of Oregon Route Views project," see http://www.antc.uoregon.edu/route-views/.

[3] "The MRT project," see http://mrt.sourceforge.net/.