# Chapter II

# Monitoring, Management And Configuration Of Networks And Networking Devices

# A Critical View Of The Sensitivity Of Transit ASs To Internal Failures

Steve Uhlig* and Sébastien Tandel**

Computing Science and Engineering Department
Université catholique de Louvain, Belgium
E-mail:{suh,sta}@info.ucl.ac.be

*(Invited Paper)*

**Abstract.** Recent work on hot-potato routing [1] has uncovered that large transit ASs can be sensitive to hot-potato disruptions. Designing a robust network is felt as overly important by transit providers as paths crossed by the traffic have both to be optimal and reliable. However, equipment failures and maintenance make this robustness non-trivial to achieve. To help understanding the robustness of large networks to internal failures, [2] proposed metrics aimed at capturing the sensitivity of ASs to internal failures. In this paper, we discuss the strengths and weaknesses of this approach to understand the robustness of the control plane of large networks, having carried this analysis on a large tier-1 ISP and smaller transit ASs. We argue that this sensitivity model is mainly useful for intradomain topology design, not for the design the whole routing plane of an AS. We claim that additional effort is required to understand the propagation of BGP routes inside large ASs. Complex iBGP structures, in particular route-reflection hierarchies [3], affect route diversity and optimality but it an unclear way.

**Keywords:** network design, sensitivity analysis, control and data planes, BGP, IGP.

## 1 Introduction

Designing robust networks is a complex problem. Network design consists of multiple, sometimes contradictory objectives. This problem has been fairly discussed in the literature, in particular [4, 5]. Examples of desirable objectives during network design are minimizing the latency, dimensioning the links so as to accommodate the traffic demand without creating congestion, adding redundancy so that rerouting is possible in case of link or router failure and, finally, the network must be designed at the minimum cost. Recent papers have shown that large transit networks might be sensitive to internal failures. In [1], Teixeira et al. have shown that a large ISP network might be sensitive

* Corresponding author. Steve Uhlig is "chargé de recherches" with the FNRS (Fonds National de la Recherche Scientifique, Belgium). This research was partly carried while Steve Uhlig was visiting Intel research Cambridge.

** Sébastien Tandel is funded by a grant from France Télécom.

to hot-potato disruptions. [6] extended the results of [1] by showing that a large tier-1 network can undergo significant traffic shifts due to changes in the routing. To measure the sensitivity of a network to hot-potato disruptions, [2] has proposed a set of metrics that capture the sensitivity of both the control and the data planes to internal failures inside a network.

To understand why internal failures are critical in a large transit AS, it is necessary to understand how routing in a large AS works. Routing in an Autonomous System (AS) today relies on two different routing protocols. Inside an AS, the intradomain routing protocol (OSPF [7] or ISIS [8]) computes the shortest-path between any pair of routers inside the AS. Between ASs, the interdomain routing protocol (BGP [9]) is used to exchange reachability information. Based on both the BGP routes advertised by neighboring ASs and the internal shortest paths available to reach an exit point inside the network, BGP computes for each destination prefix the "best route" to reach this prefix. For this, BGP relies on a "decision process" [10] to choose its a single route called the "best route among several available ones. The "best route" can change for two reasons. Either the set of BGP routes available has changed, or the reachability of the next-hop of the route has changed due to a change in the IGP. In the first case, it is either because some routes were withdrawn by BGP itself, or that some BGP peering with a neighbor was lost by the router. In the second case, any change in the internal topology (links, nodes, weights) might trigger a change in the shortest path to reach the next hop of a BGP route. In this paper we consider only the changes that consist of the failure of a single node or link inside the AS, not routing changes related to the reachability of BGP prefixes.

Our experience with the metrics proposed in [2] provided insight into the sensitivity of the network to internal failures. However, having used this sensitivity analysis on a large tier-1 network also revealed weaknesses of this model to understand the routing plane of transit ASs. We discuss further work required to help operators to understand the control plane of their network, particularly the behavior of iBGP.

Section 2 first presents the methodology to model an AS. Section 3 then introduces the sensitivity model to internal failures. Section 4 discusses the limitations of the model for realistic iBGP structures and Section 5 concludes and discusses further work in the area.

## 2 Methodology

In this section, we describe our approach to build snapshots of real ISP networks. The main point of our relatively heavy methodology is to make the model as easy as possible to match with the context of real transit ASs. We do not make assumptions on the internal graph of the iBGP sessions, even though in the case of GEANT there is a iBGP full-mesh between all border routers. Most large transit ASs do rely on route-reflection, hence putting assumptions on the sensitivity model restricts its applicability. The route solver on which we rely, C-BGP [11], has no restriction on the structure of the iBGP sessions inside an AS. C-BGP has been designed to help the evaluation of changes to the design of the BGP routing inside an AS [12]. Changes to the routing policies of an AS, or the internal configuration of its iBGP sessions is easy with C-BGP.

In this section we describe the methodology we use to model a transit AS. A more detailed discussion on how to model an AS with C-BGP can be found in [12]. We explain in this section how we build snapshots of the routing and traffic matrix of large ASs. The main steps of our methodology consist in producing snapshots of the routing inside a transit AS, consider that this view of the routing is valid for the whole bin between two time intervals, and based on the routing snapshot and the traffic statistics available to build a traffic demand for the considered time bin. Building more precise views of the routing and traffic matrices is possible by using small enough time intervals.

### 2.1 Related work

The most closely related works from the literature are [13] and [14]. The aim of [13] was to provide the networking industry with a software system to support traffic measurement and network modeling. This tool is able to model the intradomain routing and study the implications of local traffic changes, configuration and routing. [13] does not model the interdomain routing protocol though. [14] proposed a BGP emulator that computes the outcome of the BGP route selection process for each router in a single AS. This tool does not model the flow of the BGP routes inside the AS, hence it does not reproduce the route filtering process occurring within an AS. Finally, [12] discusses the problem of modeling the routing of an autonomous system and presents an open-source route solver aimed at allowing the study of networks with a large number of BGP routers.

### 2.2 Modeling the routing of an AS with C-BGP

To build snapshots of the routing inside a transit AS, we rely in this paper on C-BGP [11]. C-BGP is an open-source routing solver aimed at reproducing precisely BGP and making possible the modelling of networks containing a large number of BGP routers. The reason for choosing C-BGP instead of simply gathering BGP RIBs is that we aim at providing a tool that allows a network operator to build "what-if" scenarios, for instance by changing its topology or routing policies, and investigating their impact on the sensitivity of their network [12]. By relying on C-BGP, one can relatively simply change the internal network topology, the configuration of the routers, or even filter the set of BGP routes available inside the network.

To model a transit network with C-BGP, we use the following steps. These steps reflect the typical way we see how modeling the routing of an AS would be done, even though on particular network instances these steps might have to be slightly changed, depending on the available routing data. First, we infer the topology of the network based on its ISIS data. We create the internal POP-level topology and for each time bin, we read all the routing messages received during the time bin and inject them into C-BGP. For ISIS, we apply all link state packets (LSPs) so that both the reachability and the IGP weights at the end of the time bin are consistent with them. Note that replaying the events at their exact time is useless in C-BGP as the simulator is not even-driven, i.e. it only propagates the routing messages and lets BGP converge inside the BGP router. When only a single solution towards which the real BGP converges, C-BGP will find

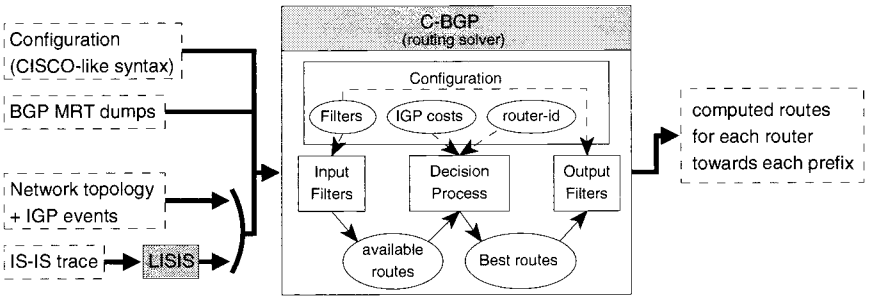



**Fig. 1.** C-BGP routing solver.

the same solution. However, when several distinct solutions exist in the real BGP [15], then it is unclear how C-BGP will behave. Sometimes is will converge towards of the the solutions, sometimes it will never converge. The model used by C-BGP provides scalability since it prevents from having to care with the computationally consuming state machines and timers of the real routing protocols.

We first inject the static topology of the transit network into C-BGP. This topology contains the routers, links, and IGP weights of the links. The shortest IGP paths inside each AS are computed at each router and BGP sessions are established between the routers. The ISIS data of the two networks up to the beginning of the actual time the simulation is supposed to start are then read and the LSPs converted into changes of the C-BGP topology. The IGP changes are injected into C-BGP and the shortest paths are recomputed at all routers.

For BGP, we restrict the set of prefixes to those largest ones, in a similar way to [16, 13, 17]. We infer by parsing the BGP RIBs where in the network the prefixes are announced by external peers. Depending on whether the transit AS relies on "next-hop-self" in its routers or not, the visibility of the eBGP routes might be different inside the iBGP sessions of the AS. It is important to understand that relying on BGP messages collected through iBGP sessions as often done in the literature implies that one does not know neither all the messages advertised by the peer routers of the network (eBGP), nor the full diversity of the BGP routes known by the routers. With the current data available today, one can only infer the state of the best routes of some router at some point in time given that it takes some time for messages to propagate inside iBGP. While this might be acceptable in some situations, not having the same set of routes as is available inside the BGP routers implies that simulating the failure of a router inside the network that is used as egress point by another router cannot be accurately simulated since the latter router will choose another best BGP route that one does not know based on the available BGP routing data.

## 3 Network sensitivity to internal failures

In this section and the following, we describe the main points of our version of the model proposed in [2]. A more detailed presentation of this model can be found in [18].

Let $G = (V, E, w)$ be a graph, $V$ the set of its vertices, $E$ the set of its edges, $w$ the weights of its edges. A graph transformation $\delta$ is a function $\delta : (V, E, w) \rightarrow (V', E', w')$ that deletes/adds a vertex or an edge from $G$. In this paper we consider only the graph transformations $\delta$ that consist in removing a single vertex or edge from the graph. In practice, an AS may want to consider more complex failures corresponding to failures that are occur together for instance, or to consider only the failures that have actually occurred in the network for the last few weeks or months. For consistency with [2], we denote the set of graph transformations of some class (router or link failures) by $\Delta G$. The new graph obtained after applying the graph transformation $\delta$ on the graph $G$ is denoted by $\delta(G)$. In this paper, we restricted the set of graph transformations as well as the definition of a graph compared to [2], as we do not consider the impact of changes in the IGP cost. Changes to the IGP cost occur rarely in real networks.

To perform the sensitivity analysis to graph transformations, one must first find out for each router how graph transformations impact the egress point it uses towards some destination prefix $p$. The set of considered prefixes is denoted by $P$. The BGP decision process $dp(v, p)$ is a function that takes as input the BGP routes known by router $v$ to reach prefix $p$, and returns the egress point corresponding to the best BGP route. The *region index set* $RIS$ of a vertex $v$ records this egress point of the best route for each ingress router $v$ and destination prefix $p$, given the state of the graph $G$: $RIS(G, v, p) = dp(v, p)$.

We introduced the state of the graph $G$ in the *region index set* to capture the fact that changing the graph might change the best routes of the routers. In AS that consist of a full-mesh of iBGP sessions, the impact of a node or router failure on the best route used by a particular router is straightforward to find out. Removing a node or link changes the shortest paths to reach the egress points. Simulating the decision process of the router is enough to predict the best route after the graph transformation. In more complex networks on the other hand, the failure of a link or node not only changes the shortest paths towards the egress points, but new routes might also be advertised in replacement of a previously known route. In such a case, the BGP convergence inside the AS must be replayed to know the exact outcome of the BGP decision process.

The next step towards a sensitivity model is to compute for each graph transformation $\delta$ (link or router deletion), whether a router $v$ will change its egress point towards destination prefix $p$. For each graph transformation $\delta$, we recompute the all pairs shortest path between all routers after having applied $\delta$, and record for each router $v$ whether it has changed the egress point for its best BGP route towards prefix $p$. We denote the new graph after the graph transformation $\delta$ as $\delta(G)$. As BGP advertisements are made on a per-prefix basis, the best route for each $(v, p)$ pair has to be recomputed for each graph transformation. It is the purpose of the *region shift function* $H$ to record the changes in the egress point corresponding to the best BGP route of any $(v, p)$ pair, after a graph transformation $\delta$:

$$H(G, v, p, \delta) = \begin{cases} 1, \textit{if } RIS(G, v, p) \neq RIS(\delta(G), v, p) \\ 0, \textit{otherwise} \end{cases}$$

The *region shift function* $H$ is the building block for the metrics that will capture the sensitivity of the network to the graph transformations.

To summarize how sensitive a router might be to a set of graph transformations, the *node sensitivity* $\eta$ computes the average *region shift function* over all graph transformations of a given class (link or node failures), for each individual prefix $p$:

$$\eta(G, \Delta G, v, p) = \sum_{\delta \in \Delta G} H(G, v, p, \delta) \cdot Pr(\delta)$$

where $Pr(\delta)$ denotes the probability of the graph transformation $\delta$. Note that we assume that all graph transformations within a class (router or link failures) are equally likely, i.e. $Pr(\delta) = \frac{1}{|\Delta G|}, \forall \delta \in \Delta G$, which is reasonable unless one provides a model for link and node failures. Further summarization can be done by averaging the *vertex sensitivity* over all vertices of the graph, for each class of graph transformation. This gives the *average vertex sensitivity* $\hat{\eta}$:

$$\hat{\eta}(G, \Delta G, p) = \frac{1}{|V|} \sum_{v \in V} \eta(G, \Delta G, v, p)$$

The *node sensitivity* is a router-centric concept that performs an average over all possible graph transformations, measuring how much a router will change its egress point for its best routes after the graph transformations on average. Another viewpoint is to look at each individual graph transformation $\delta$ and measure how it impacts all routers of the graph. The *impact of a graph transformation* $\theta$ is computed as the average over vertices of the *region shift function*:

$$\theta(G, p, \delta) = \frac{1}{|V|} \sum_{v \in V} H(G, v, p, \delta)$$

The *average impact* of a graph transformation $\hat{\theta}$ summarizes the information provided by the *impact* by averaging it over all graph transformations of a given class:

$$\hat{\theta}(G, \Delta G, p) = \sum_{\delta \in \Delta G} \theta(G, p, \delta) \cdot Pr(\delta)$$

## 4 Discussion

Our use of the sensitivity model sketched in section 3 on different networks has shown the interest and limitations of this model. The first thing to be noted is that the model is very "hot-potato centric". For networks that do not have a complex iBGP structure (no route-reflection), the model reveals the critical links and routers [18]. That is, the model captures the sensitivity due to the concentration of many internal paths that will change after a graph transformation. However, using the model for complex transit networks using route-reflection is more tricky. Contrary to the case of an AS with an iBGP full-mesh, route-reflection introduces opacity into the selection of the best BGP route by a router. In the case of an iBGP full-mesh, each ingress router, i.e. a router that is not itself an exit point inside the AS for the considered destination prefix, chooses as its best route the one that is the "best" one among all the routes advertised by the external peers of the AS. If the routing policies applied inside the AS are consistent among all

internal routers, then for each ingress router there exists only one best BGP route that will be chosen by this router, and the choice of this route will not depend on the best route choice of the other routers of the AS. In the case of route-reflection, the best route chosen by a router depends on the best route choice of the route reflectors on the BGP routes propagation path inside the AS.

When an iBGP full-mesh is used, it is relatively straightforward to predict the outcome of an internal change on the best route choice performed by BGP. Among several alternative routes having the same quality for BGP[1], the cost of the IGP path to reach the exit point inside the AS will be used to decide which BGP route will be considered as best. Hence in an iBGP full mesh, there is a direct relationship between the IGP shortest paths and the best route choice made by BGP. For instance, Figure 2 shows the internal topology of a simple transit AS relying on an iBGP full-mesh. Inside the iBGP, BGP routers only propagate to other iBGP peers routes they did not receive from an iBGP peer. We do not show all the iBGP sessions on Figure 2, but all routers have an iBGP session with every other router of the topology. We also consider a single destination prefix $p$. The transit AS has three ingress routers $i1$, $i2$, and $i3$, two egress routers $e1$ and $e2$, and a router located in the middle of the topology. All routers run both the IGP and BGP as in most ASs. The arrows on Figure 2 provide the choice of the best route by BGP towards $p$ at each router of the AS. On Figure 2, two eBGP routes are learned to reach prefix $p$, one through egress router $e1$ and another through egress router $e2$. With an iBGP full-mesh, $i1$ and $i2$ will use the BGP route learned through $e1$, while $i3$ will use the BGP route learned through $e2$, both due to the IGP cost rule of the decision process. Each router hence relies for its best BGP route the smallest IGP cost path to exit the network.

Now suppose that our AS relies on route-reflection [3] as shown on Figure 3. The sole difference between Figure 2 and Figure 3 in terms of the routing configuration concerns the iBGP sessions. With route-reflection, all BGP routers are not directly connected anymore by an iBGP session, but all border routers are clients of the route-reflector RR. In this case, each routers knows the routes it learned from eBGP sessions, and a single route advertised by RR. The issue with route-reflection is that the best route chosen by RR depends on its own IGP cost to reach the exit point inside the AS, not the one of its clients. RR will choose as its best BGP route the one learned through $e1$ since its IGP cost is smaller than the one through $e2$. In that case, $i1$, $i2$ and $i3$ will all choose as their best route to reach $p$ the route advertised by RR, and thus will use $e1$ as their exit point towards $p$. The IGP cost of the best route chosen by $i3$ is not optimal in terms of the IGP cost anymore compared to the iBGP full-mesh. Here we used a simple situation with a single route-reflector. In practice, large transit ASs can rely on a hierarchy of route-reflectors. Route-reflection trades-off the number of iBGP sessions inside the AS with a drastic reduction in the diversity of the routes known by the routers, and by a potentially suboptimal IGP cost of the best routes chosen by the routers.

A route reflector today chooses for all its clients routers a single best BGP route. The impact of this choice on the client routers is that the latter's will be sensitive to the graph transformations that affect the path of this best route chosen by their route reflector. Depending on how these client routers are located inside the AS, a particular

[1] Same value of local-pref, AS path length, MED, and route origin type.

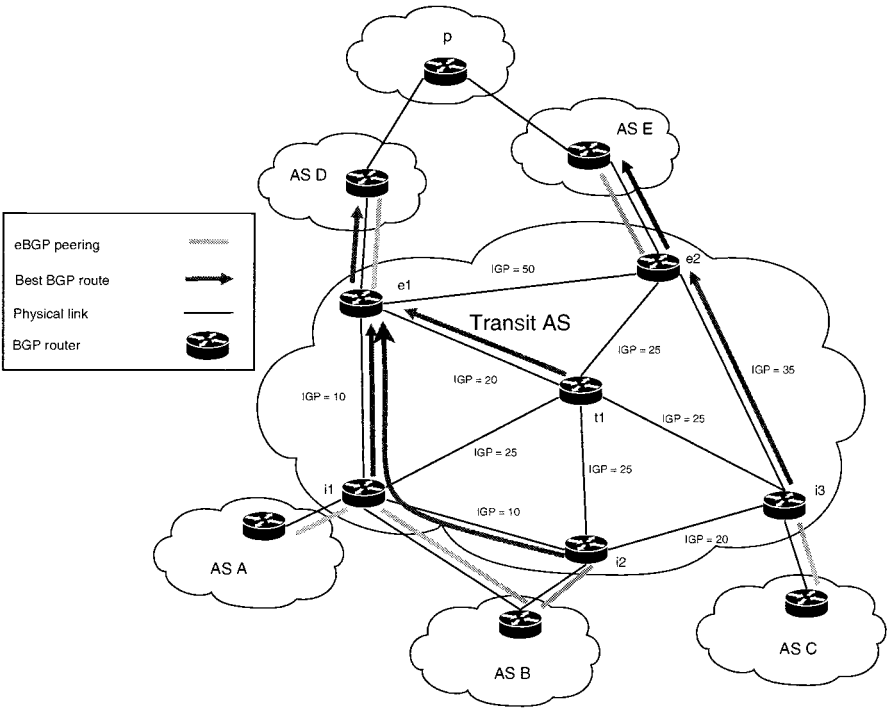



**Fig. 2.** Example topology with iBGP full-mesh.

subgraph of the AS will be used by the clients to reach the egress point of the BGP route advertised by the route reflector.

The implication for the network sensitivity is that the impact of the graph transformations and the node sensitivity will depend very much on how route-reflectors redistribute the routes they learn to their clients, and which routes they consider as best. If each route reflector was to compute for each of its client the latter would have selected in the case of an iBGP full-mesh and redistribute this route on a per-client basis as proposed in [19], then the sensitivity of the network in the two cases would be the same. However, if route reflectors are left to redistribute their best route without respect to how their clients would have chosen their best route in the iBGP full-mesh, then it is difficult to foresee how this will change the paths used to carry the traffic inside the AS since it depends both on the placement of the route reflectors, the interconnection structure of the iBGP sessions, the diversity of the BGP routes learned from peer ASs, and wherefrom the external routes are learned.

Our experience with a large tier-1 network have shown that the results of the sensitivity analysis between the actual network configuration and an iBGP full mesh are quantitatively much different but qualitatively very alike. However, understanding why differences in the sensitivity arise when the iBGP structure is changed is not answered by the sensitivity model of [2]. For that, one must first understand what filtering is

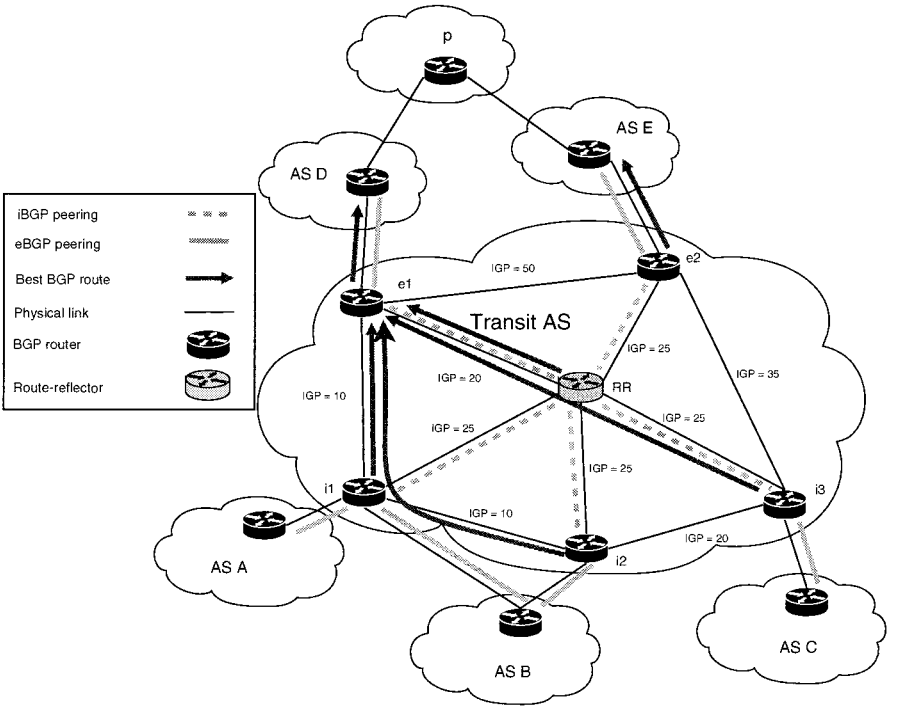



**Fig. 3.** Example topology with route reflection.

performed by a given iBGP structure compared to the iBGP full mesh. The reason to compare with an iBGP full mesh is that the best route visibility is achieved by the full mesh, while introducing route reflection reduces the diversity of the routes that can be chosen by the BGP routers inside the AS.

## 5 Conclusions and further work

In this paper we have presented our version of the sensitivity model to internal failures based on the work of [2]. We sketched our methodology to study the sensitivity of an AS based on this model and discussed the limitations of this model to provide insight into the behavior of the control plane of large transit ASs due to the presence of route reflectors.

The obvious further work we see is to study the actual impact of route reflection on the diversity and sensitivity of the routers inside a large transit AS. We are currently investigating this problem on a large tier-1 network. The lack of study in the literature concerning the actual impact of route reflection on the diversity and the filtering of the routes inside an AS indicates that our current understanding of iBGP is still very poor.

We feel that a proper understanding of iBGP is particularly important for the design of robust networks.

## Acknowledgments

This research was partially supported by the Walloon Government (DGTRE) within the TOTEM project [20]. We thank all the people from DANTE that helped in making the GEANT routing and traffic data available, and among them Nicolas Simar specifically. We would also like to thank Bruno Quoitin for developping C-BGP [11].

## References

1. R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford, "Dynamics of hot-potato routing in IP networks," in *Proc. of ACM SIGMETRICS*, June 2004.
2. R. Teixeira, T. Griffin, G. Voelker, and A. Shaikh, "Network sensitivity to hot potato disruptions," in *Proc. of ACM SIGCOMM*, August 2004.
3. T. Bates, R. Chandra, and E. Chen, "BGP Route Reflection - An Alternative to Full Mesh IBGP," Internet Engineering Task Force, RFC2796, April 2000.
4. R. S. Cahn, *Wide Area Network Design: Concepts and Tools for Optimisation*, Morgan Kaufmann, 1998.
5. W. D. Grover, *Mesh-Based Survivable Networks*, Prentice Hall PTR, 2004.
6. R. Teixeira, N. Duffield, J. Rexford, and M. Roughan, "Traffic matrix reloaded: impact of routing changes," in *Proc. of PAM 2005*, March 2005.
7. J. Moy, *OSPF : anatomy of an Internet routing protocol*, Addison-Wesley, 1998.
8. D. Oran, "OSI IS-IS intra-domain routing protocol," Request for Comments 1142, Internet Engineering Task Force, Feb. 1990.
9. J. Stewart, *BGP4 : interdomain routing in the Internet*, Addison Wesley, 1999.
10. Cisco, "BGP best path selection algorithm," `http://www.cisco.com/warp/public/459/25.shtml`.
11. B. Quoitin, "C-BGP, an efficient BGP simulator," `http://cbgp.info.ucl.ac.be/`, September 2003.
12. B. Quoitin and S. Uhlig, "Modeling the routing of an Autonomous System with C-BGP," *IEEE Network Magazine*, November 2005.
13. Anja Feldmann, Albert Greenberg, Carsten Lund, Nick R eingold, and Jennifer Rexford, "NetScope: Traffic Engineering for IP Networks," *IEEE Network Magazine*, March 2000.
14. N. Feamster, J. Winick, and J. Rexford, "A model of BGP routing for network engineering," in *Proc. of ACM SIGMETRICS*, June 2004.
15. T. Griffin and G. Wilfong, "An analysis of BGP convergence properties," in *Proc. of ACM SIGCOMM*, September 1999.
16. A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational IP networks: methodology and experience," in *Proc. of ACM SIGCOMM*, September 2000.
17. J. Rexford, J. Wang, Z. Xiao, and Y. Zhang, "BGP Routing Stability of Popular Destinations," in *Proc. of ACM SIGCOMM Internet Measurement Workshop*, November 2002.
18. S. Uhlig, "On the sensitivity of transit AShs to internal failures," in *Proc. of the fifth IEEE International Workshop on IP Operations and Management (IPOM2005), Barcelona, Spain*, October 2005.

19. O. Bonaventure, S. Uhlig, and B. Quoitin, "The case for more versatile BGP Route Reflectors," Work in progress, draft-bonaventure-bgp-route-reflectors-00.txt, July 2004.
20. "TOTEM: a TOolbox for Traffic Engineering Methods," `http://totem.info.ucl.ac.be/`.

# The CoMo Project: Towards A Community-Oriented Measurement Infrastructure

Gianluca Iannaccone and Christophe Diot

Intel Research, Cambridge, UK

*(Invited Paper)*

**Abstract.** CoMo (Continuous Monitoring) is a passive monitoring system. CoMo has been designed to be the basic building block of an open network monitoring infrastructure that would allow researchers and network operators to easily process and share network traffic statistics over multiple sites. This paper identifies the challenges that lie ahead in the deployment of such an open infrastructure. These main challenges are: (1) the system must allow any arbitrary traffic metric to be extracted from the packet streams, (2) it must provide privacy and security guarantees to the owner of the monitored link, the network users and the CoMo users, and (3) it must be robust to anomalous traffic patterns or traffic query loads. We describe the high-level architecture of CoMo and, in greater detail, the resource management, query processing and security aspects.

**Keywords:** Network measurements and monitoring.

## 1 Introduction

Despite the great interest of recent years in measurement based Internet research, the number and the variety of data sets and network monitoring viewpoints remains unacceptably small. Several players in the network measurement area, like CAIDA [2], NLANR [18], RouteViews [23], RIPE [21], Internet2 [13] and, recently, GEANT [9] have provided data sets with various degrees of accuracy and completeness. Some large commercial ISPs also deploy private infrastructures and share limited information with the rest of the research community [6, 8]. Other network operators (e.g., corporate networks, stub ISPs, Universities) instead tend to lack a measurement infrastructure or, in case they do have one, do not share any data or even report the existence of such infrastructure.

This situation constitutes a major obstacle for network researchers. It hampers the ability to validate the results over a wide and diverse range of datasets. It makes it difficult to generalize results and identify the presence of traffic characteristics that are invariant and common to all networks. For example, it is difficult to quantify the magnitude of a denial of service attack or a worm infection, to evaluate the relevance of network pathologies such as in-network packet

duplication and reordering, or to simply identify the dominant applications in the Internet.

So far, several barriers have limited the ability of researchers to deploy and share large number of network datasets:

- The cost of the monitoring infrastructure that should be present on a large set of links with speeds varying from the few Mbps of small organizations up to the 10Gbps of ISPs' networks.
- The lack of software tools for managing a passive monitoring infrastructure. Today, monitoring systems use ad-hoc tools that are not appropriate for large infrastructures. For example, they tend to provide poor and inefficient query interfaces.
- The lack of a standard open interface to access the monitoring system. Although, several efforts exist to provide a single packet trace format (e.g., tcpdump/libpcap [22], IETF IPFIX working group [14]), no standard exists to access high speed network monitoring devices.
- The difficulty in controlling the access to the data avoiding to disclose private information about network users or organizations.

We have designed CoMo, an open software platform for passive monitoring of network links. CoMo has been designed to be flexible, scalable, and to run on commodity hardware. With CoMo, we intend to lower the barriers described above and encourage the deployment of a large scale monitoring infrastructure. We believe this effort constitutes a necessary first step towards a better understanding of network protocols and traffic. For example, the extensible nature of CoMo allows early deployment of novel methods for traffic analysis, anomaly diagnosis or network performance evaluation.

The rest of the paper describes the challenges posed by the design of the CoMo platform, its resulting architecture, and some specific aspects of CoMo such as the query engine, resource management and security.

## 2 Challenges

The design of a system such as CoMo is driven by the trade-off between openness and resilience. The system should be open enough to allow authorized users to compute allowed metrics on the observed traffic. At the same time, it should allow link owners to specify constraints on how the collected data can be used. Finally, the system should be robust to abuses by unauthorized users, and have a predictable and graceful performance degradation in periods when it cannot sustain the incoming packet rate, We can summarize the system requirements as follows:

**Openness.** The users should be able to customize the system and the software platform to their specific needs and deployment environment. For example, a user interested in intrusion detection or performance analysis may only need limited storage; on the other hand, an ISP's network operator interested in post-mortem

analysis might require to store and retrieve a large and detailed packet-level or flow-level information.

The metrics also need to be dynamically configurable in order to address a large range of applications such as network trouble-shooting, anomaly and intrusion detection, SLA computation, etc. In addition, the system should allow users to easily interact and interface with it in order to start or stop some metric computation or to run a query on the collected datasets.

**Resilience.** This requirement is often orthogonal to the previous one. First and most important, the system should be able to monitor *and* analyze the traffic in any load condition, and in particular in the presence of unexpected traffic anomalies that may overload the system resources. The system should control its resources carefully. For example, the computation of one metric should not monopolize the use of resources, starving other crucial system tasks (e.g., packet trace collection). The owner of the system needs to be able to control the access to the system. Various type of users will want to access a monitoring system, including malicious ones. Because of its exporting capabilities, a system can impact the network it measures. Different request will be processed with different priorities. The system should also make sure it does not compute the same metric twice for two different users or applications.

### 2.1 Design Challenges

Given the requirements described above, we identify four main design challenges:

**Ease of deployment.** The success of the CoMo infrastructure will be a function of how simple it is for user to access the infrastructure, specify and implement traffic metrics and analysis methods, query the data from the system. As we will point out in Section 3, many of the design decisions are driven by the need to trade architecture simplicity for efficiency and performance.

**Query interface.** Designing a query interface with the constraints defined in the requirement section poses two problems: (*i*) how to express the query; (*ii*) how to run the query without explicit built-in support into the system. Expressing a query may be particularly hard given that the system is supposed to be used by a large range of users, defining new traffic metrics and analysis methods. It is unlikely that all metrics will be specified well enough to be translated in a standard query language; metrics may require new constructs that are not present in the original query language. However, the system should still allow custom-built queries to run. We will address this problem in Section 4.

**Resource Management.** Opening the system to a potentially large number of users requires a very careful resource management, i.e. CPU, memory, I/O bandwidth or storage space. Indeed, allowing users to compute any metric on the traffic stream may result in analysis that are particularly computing intensive, and in a large amount of data to be exported. Therefore, the system needs to define strict policies for analysis metrics and needs to be able to enforce them

and possibly to adapt based on the load of the system. We will address the problem of resource management in Section 5.

**Security issues.** CoMo users will have different rights on the system depending also on the system environment. For example, a network operator may be allowed to inspect the entire packet payload in order to spot viruses or worms, while a generic user may only be able to access the packet header (probably anonymized). A second security aspect is related to the vulnerability of the monitoring system. CoMo systems contain confidential information. They can also export large amounts of traffic and impact the network where they are installed. An attacker may also target directly the CoMo system by preventing users to access the system or by corrupting the collected data. In Section 6 we will describe the threat model for CoMo and propose initial solutions.

## 3 Architecture

This section presents a high-level description of the architecture and an overview of the major design choices. We call *data* any measurement related information. Data include original packets captured on the monitored links, as well as statistics computed on the packets and other representations of the original packet trace such as flow records.

### 3.1 High level architecture

The system is made of two principal components. The *core processes* control the data path through the CoMo system, including packet capture, export, storage, query management and resource control. The *plug-in modules* are responsible for various transformations of the data.

The data flow across the CoMo system is illustrated in Figure 1. The white boxes indicate plug-in modules while gray boxes represent the core processes. On one side, CoMo collects packets (or subsets of packets) on the monitored link. These packets are processed by a succession of core processes and end stored onto hard disks. On the other side, data are retrieved from the hard disk on user request (by the way of queries addressed to a CoMo system). Before being exported to users, those data go through an additional processing step.

As explained earlier, the modules execute specific tasks on data. The core processes are responsible for the "management" operations, common to all modules (e.g., packet capture and filtering, data storage). The following tasks also fall under the responsibility of the core component: (*i*) resource management, i.e., deciding which plug-in modules are loaded and running, (*ii*) policy management to manage the access privileges of the modules, (*iii*) on-demand requests handling to schedule and respond to user queries, and finally (*iv*) exception handling to manage the situation of traffic anomalies and the possible graceful degradation of system performance.

The modules take data on one side and deliver user-defined traffic metrics or processed measurement data on the other side. One of the challenges identified

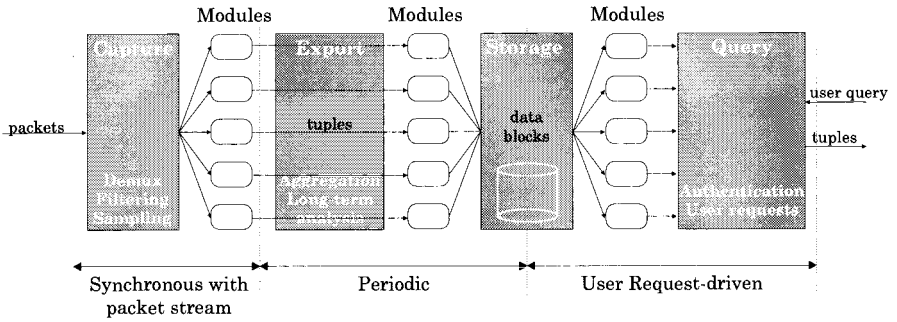



**Fig. 1.** Data flow in the CoMo system

in Section II is to keep modules very simple. All complex functions should be implemented within the core component. This strict division of labor allows us to optimize the core component[1], while the modules can run sub-optimally and can be implemented independently by any CoMo user.

### 3.2 The core processes

The core processes are in charge of data movement operations (i.e., from the packet capture card to memory and to the disk array). Moving data in a PC is the most expensive task given memory, bus and disk bandwidth limitations. Therefore, in order to guarantee an efficient use of the resources, it is better to maintain a centralized control of the data path. However, one of the goal of the architecture is to allow the deployment of CoMo as a cluster using dedicated hardware systems (such as network processors) for high performance monitoring nodes.

Communication between core processes is governed by a unidirectional message passing system to enable the partition of functionality over a cluster.

In a single system, a CoMo node uses shared memory and Unix sockets for the signaling channel. The use of processes instead of threads is justified by the need of high portability of the software over different operating systems.

Two basic guidelines have driven the assignment of the functionalities among the various processes. First, functionalities with stringent real-time requirements (e.g., packet capture or disk access) are confined within a single process (*capture* and *storage*, respectively). The resources assigned to these processes must be able to deal with worst case scenarios. Other processes instead operate in a best-effort manner (e.g., *query* and *supervisor*) or with less stringent time requirements (e.g., *export*). Second, each hardware device is assigned to a single process. For example, the *capture* process is in charge of the network sniffer, while *storage* controls the disk array.

[1] The CoMo code is open source and we aim to build an open community of developers in charge of the core components.

Another important feature of our architecture is the decoupling between real-time tasks and user driven tasks. This is visualized by the vertical lines in Figure 1. This decoupling allows us to control more efficiently the resources in CoMo and to avoid that a sudden increase in traffic starves query processing, and vice-versa.

We now describe the five main processes that compose the core of CoMo:

- The *capture* process is responsible for the packet capture, filtering, sampling and maintaining per-module state information;
- The *export* process allows long term analysis of the traffic and provides access to additional networking information (e.g., routing tables);
- The *storage* process schedules and manages the accesses to the disks;
- The *query* process receives user requests, applies the query on the traffic (or reads the pre-computed results) and returns the results;
- The *supervisor* process is responsible for handling exceptions (e.g., process failures) and to decide whether to load, start or stop plug-in modules depending on the available resources or on the current access policies.

The *capture* process receives packets from the network card (that could be a standard NIC card accessed via the Berkeley Packet Filter [16], or using dedicated hardware such an Endace DAG card [7]). The packets are passed through a filter that identifies which modules are interested in processing the packets. Then the *capture* process communicates with the modules to have them process the packets and update their own data structures. Note that those data structure may also be maintained by the *capture* process in order to keep the module simple.

Periodically, *capture* polls the data structures updated by the modules and sends its content to the *export* process. These data structures are then ready to be processed again by the modules. As explained earlier, this way, we decouple real-time requirements of the *capture* process that deals with incoming packets at line rate from storage and user oriented tasks. Flushing out the data structure periodically also allows *capture* to maintain limited state information and thus reduce the cost of insertion, update and deletion of the information stored by the modules.

The *export* process mimics the behavior of *capture* with the difference that *export* handles state information rather than incoming packets. Therefore, *export* communicates with the modules to decide how to handle the state information. A module can request *export* to store the information and/or to maintain additional, long term information. Indeed, as opposed to *capture*, *export* does not flush periodically its data. Instead, it needs to be instructed by the module to get rid of any data.

The *storage* process takes care of storing *export* data to the hard disk. The *storage* process is data agnostic and treats all data blocks equally[2]. It can thus

[2] A viable alternative is to allow *storage* to filter some of the data blocks as early as possible to reduce data movement and processing, following an approach similar to Diamond [12].

focus on an appropriate scheduling of disk accesses and on managing disk space. The *storage* process understands only two type of requests: "store" requests from *export* and "load" requests from *query*.

The *query* process manages users' requests, and if access is granted to the user, it gets the relevant data from disk via the *storage* process and returns the query results to the user. If the requested data is not available on disk, the *query* process can (*i*) perform the analysis on the packet trace stored on the disk (if the request refers to a period in the past) or (*ii*) request the initialization of a new module by the *supervisor* process to perform the query computation on the incoming packet stream.

Finally, the *supervisor* monitors the other processes and decides which modules can run based on the available resources, access policies and priorities. The *supervisor* communicates with all the other processes to share information about the overall state of the system.

### 3.3 Plug-in Modules

The traffic metrics and statistics are computed within a set of plug-in modules. The modules can be seen as a pair *filter:function*, where the filter specifies the packets on which the function should be performed. For example, if the traffic metric is "compute the number of packets destined to port 80", then the filter would be set to capture only packets with destination port number 80, while the function would just increment a counter per packet. Note that all modules do not necessarily compute statistics. Modules can simply transform the incoming traffic, like for example transform a set of packets in a flow record.

The core processes are responsible for running the packet filters and communicate with the modules using a set of callback functions. Actually there are several sets of callback functions, one for each of the core processes (represented by the three columns of white boxes in Figure 1). Going back to the previous example, the *capture* process will use a callback (`update()`) to let the module increment the counter. Then the *export* process will use a different callback (`store()`) to move the counter value to the disk. Also, *export* could use another callback to allow the module to apply, for example, a low pass filter on the counter values. The *Query* process will then use a different callback (`load()`) to retrieve the counter value from disk.

It is important to observer that the core processes are agnostic to the state that each module computes. Core processes just provide the packets to the module and take care of scheduling, policing and resource management tasks.

## 4 Querying Network Data

The query engine is the CoMo gateway to the rest of the world. The main function of queries is to request CoMo to export data. The range of data to be exported can vary significantly, from raw packet sequences to aggregated traffic statistics.

The processing of a query can be divided in three steps: (*i*) validate and authorize the query (as well as its origin), (*ii*) find and/or process the data and, finally, (*iii*) send the data back to the requester.

The amount of data stored on a CoMo system can be very large (in the order of 1TB on current prototypes). It is thus desirable to reduce the amount of processing needed to answer a query to a minimum. This is, indeed, the main purpose of the CoMo modules: pre-compute data to minimize the cost of processing incoming queries.

In CoMo we identify three types of queries:

- *Triggers* defined in the system configuration together with the relevant module. This kind of query will appear in the form `'send-to <IP address>:<port>'` and follow a push information model, i.e. as the module computes the metric on the traffic stream, data is sent to the specified IP address to trigger new computation or an alert.
- *On-demand queries* explicitly specify the relevant module. This could happen in two ways: indicating the name of the module in the query itself or sending directly the module source code. The query would then have to indicate the packet filter to be applied to the packet stream and the time window of interest. The response consist in the output of the module. On reception of this query, the CoMo system has to authenticate the module and the requester, and then figure out if the same module has already been installed. If the same module has been running during the time of interest, then this query revert to a static query. Otherwise, it requires the module to run on the stored packet-level trace[3] with an obvious impact on the query response time.
- *Ad-hoc queries* have no explicit module defined. These queries are written in a specific query language and code for the modules is generated on the fly. A similar approach is followed in systems like Aurora [3], TelegraphCQ [4], Gigascope [6] or IrisNet [10]. The caveat of this kind of approach is that it cannot always exploit existing modules that have been running on the packet stream. One solution is to have each module generate a common representation of the data it has computed so that the query process can compare the received query with all the modules to find a module that is computing a *super-set* of the response. One common representation could be a packet stream that all modules are designed to interpret correctly. Therefore, each module could include routines to "regenerate" a packet stream that resembles the one originally monitored. If the partial information contained in this regenerated stream is sufficient for the new query then it can run without any modification.

One of the big challenges of the system is to manage queries both in terms of computing resources and data transfers. Processing a query should not jeopardize

[3] Note that every CoMo system is supposed to keep a packet-level trace at all times. The duration of the packet trace will depend on the available storage space and the link speed.

the ability of the system to collect the packet-level trace without losses. Thus, it is important for the system to predict the usage of resources of a query before scheduling it to run. This can be easily done for trigger queries (the module is known in advance to the system), while it is more challenging for other type of queries.

Moreover, the system should take into account the priority of queries and that of all the other modules in the system that are pre-processing data for future queries. Indeed, some queries might be urgent (e.g., real-time security related information) and it might be appropriate not to delay the computation of the query (at the cost of other CoMo tasks).

Finally, it is most probable that multiple CoMo systems will be present in a network. Consequently, more than one system may be needed to answer a query or multiple systems may coordinate to identify the most appropriate subset of them over which to run the query. For example, a system could be more appropriate than others depending on the relevance of the monitored traffic for the query, (e.g., when tracking a denial of service attack) or the current system load. Moreover, some of these systems will have data to export, some will not. Therefore, an additional challenge is to design a query management system that minimizes the search and export cost in the context of distributed queries. For this specific challenge, we will also investigate innovative solutions for query management proposed in the context of sensor networks [1, 15, ?].

## 5 Resource Management

Managing system resources (i.e. CPU, hard disk, memory) in CoMo is challenging because of two conflicting system requirements. On the one hand, the system should be open to users to add plug-in modules for new traffic metrics and to query the system. On the other hand, the system needs to be always available, guarantee a minimum performance level and compute accurately the metrics that are of interest at a given time.

We divide the major causes of resource consumption in three categories: traffic characteristics, measurement modules, and queries. In the following we will address each of them separately.

**Traffic characteristics**. Resource utilization depends heavily on the incoming traffic, affecting both the core processes, which have per-packet processing costs, and the modules, whose consumption of storage, CPU cycles and (indirectly) disk I/O usage varies depeding on traffic characteristics and the type of metrics computed.

For example, a module could be idle for long periods of time and then have a burst of activity when packets hit its filter, possibly resulting in turn in large amounts of data to be stored to disk. A module computing flow statistic would become very memory-greedy in case of DoS attack. The amount of computations per packet will often depend on the packet content, e.g. in the case of IDS's.

**Modules.** Apart from the fixed per-packet processing costs, which only depend on the input traffic flows, most of the resource utilization costs depend on the activity of the modules. To control the resources used by modules, we impose a number of constraints on their operation:

- *Modules can be started and stopped at any time.* Modules are prioritized based on resource consumption and managed accordingly. We also rely on the fact that most module can be run off-line at a later time on the packet trace.
- *Modules have access to a limited set of system calls.* They cannot allocate memory dynamically and have no direct access to any I/O device. This allows us to maintain the control over the resource usage within the core processes and, at the same time, it keeps the module's source code simple.
- *Modules do not communicate with each other.* Modules are independent. They do not share any information with other modules. This constraint simplifies resource management, although it introduces redundancies. For example, one module cannot pass pre-processed information to another module. This way, different modules might perform the same computations on the same packets.

These three module requirements allow the CoMo system to regulate the amount of computing resources used at a given time. However, the decision on whether to start or stop a module depends on two parameters: (*i*) the measured resource usage of the module and (*ii*) the relevance of the metric computed by the module. The optimization of the sets of modules run at a given time is also a challenging issue. It is very difficult to estimate the resource usage (that depends on incoming traffic) and the relevance of a given metric can also vary significantly over time.

**Queries.** Queries can come at anytime and cause resource consumption for authentication, module insertion or removal, data processing. The query engine will have to manage the impact of the query processing on the system resources and active modules. The main issue with queries is that they should have higher priority than existing modules. A query indicate that a user exists and is waiting for results, while modules are just pre-computing queries based on the assumption that some users will be interested. On the other hand, in presence of a large number of queries, running them at high priority may force the CoMo system to be a simple packet collector, reducing the usefulness of the modules.

### 5.1 Resource control options

As described above, the prediction of resource utilization is almost impossible given the different factors that affect it. Therefore, we have no other option than accurately measuring resource utilization and timely react to overload situations. We need to identify what "knobs" can be turned to control resource utilization.

In CoMo, resource management is threshold-based. The resource usage of a CoMo system is monitored continuously in term of CPU cycles, memory usage

and disk bandwidth. If the usage exceeds the pre-defined high-threshold, the following actions can be taken: (*i*) sample the incoming traffic for certain modules; (*ii*) stop some modules; (*iii*) block incoming queries.

The specific action to be taken should depend on the module priorities. As we said earlier, most modules can be delayed and run at a later time on the stored packet trace. The priority of a module should depend on the requested response time of the potential query that needs the data computed by the module. For example, a module computing a metric for anomaly detection should have high priority given that a query for anomalies requires in general a very short response time.

The module priority should then adapt to the query currently running on the system as well as on the historical queries that the system has received. At configuration time, the system administrator will define a static priority for each module that will then vary depending on how often the data processed by a module are actually read.

## 6 Security Issues

There is no doubt that the greatest challenge in providing an open monitoring service to users consists in enforcing access policies and safeguarding the privacy of network users.

One static policy applied to all systems is not enough given the large number of different uses that we envision for the monitoring infrastructure. For example, the network operator that owns the monitored link may have complete access to the traffic, including the payload. Other users should instead only be allowed to view the packet header or even just an anonymized versions of it. Finally, some queries could be limited to a subset of the users in order to avoid a constant overload of the system or to increase network traffic (e.g., all queries that require large data transfers). Hence, a rich and descriptive policy language is needed. The policy should define which modules can be plugged into the system, which users can plug modules in, and which users can post queries to the system together with the type of queries they can post.

### 6.1 Access policies

The only access points to the system for users is the plug-in interface where new modules are added and the query interface. However, note that a query always results in a module being plugged into the system. Therefore, in the rest of this section we will consider only the case of a user requesting to plug in a module.

The module is described by the two components *filter:function*. It is possible to assign *access request levels* to each component. These access requests levels indicate the privilege level at which the module has to run (and that has to match the user access privileges). For example, a filter that specifies "anonymized packet headers" will have the lowest access request level, while a filter that desires

to have access to "not anonymized packet payloads" will be marked with the highest access request level.

Assigning access requests level for the *functions* of the modules is a much harder problem. It requires a deep understanding of what the module is computing and storing to disk. The solution that is often adopted is to allow the function to perform any computation but to restrict significantly the filter. NLANR, for example, provides only anonymized packet header traces but does not impose any condition on what user do with the traces [18]. Unfortunately, this approach is not appropriate in general. For example, worm signature detection requires full inspection of the packet stream although the state information it maintains (worm traffic) has little relevance and would certainly have a low access request level.

The approach followed by CoMo is to allow to load on the system only "signed" modules, for which the original developer can be authenticated. Then the module's function will inherit the developer privileges. User access privileges will initially depend on the CoMo system itself: (*i*) public access system will allow any user to plug-in modules and query the system; (*ii*) restricted system where only a subset of the users are allowed to plug-in new modules and the rest of the users can only perform queries on metrics for which a module already exists; (*iii*) private access, where only a subset of users can plug-in modules and query the system. In the future, we envision that each user will have individual privilege levels that will decide whether a *filter:function* pair is allowed to run on CoMo.

### 6.2 Infrastructure Attacks

So far, we have only addressed the security of the data. We now discuss the possible attacks on the monitoring infrastructure. We consider two types of attacks:

**Denial of service attacks.** Attacks in this class may come in the form of a module that uses a disproportionate amount of resources or that corrupts the data owned by other modules. The former type of attacks could be dealt directly with the resource manager and the use of "module black list" to forbid a module to run again in the system. Also, the use of a sandbox or of signed modules may help in avoiding this class of attacks and finding out a module's real "intentions". In fact, because modules process incoming traffic, on which we have little if any control, even a perfectly legal module could be driven into consuming large amount of resources in response to certain input patterns. In general this problem is dealt with by the generic resource control mechanisms discussed in Section 5.

The second class of attacks (data corruption) is harder to defend against. One first immediate solution is to provide memory isolation between modules. This can be achieved running modules as separate processes or moving some of the CoMo functionalities in the kernel. This introduces some overhead on the system but would guarantee that two modules will not interfere with each other.

**Attack on the access policies.** This class of attacks include attacks on the user privileges or on the access request level of a filter or function. For example, one can envision an attack on the packet anonymization scheme that would allow a user with low privileges to run a filter with high access level. An attack on the anonymization scheme could consist in sending carefully-crafted packets to the system and use a module that captures the anonymized version of those packets in an attempt to break the anonymization scheme [24].

## 7 Related Work

The list of software and techniques for active monitoring and network performance metrics computation is long. NIMI [20] is the pioneer in the deployment of active monitoring infrastructures, while CAIDA [2] has made available a large set of tools for active monitoring.

The area of passive monitoring is much less rich than active monitoring, mostly because of the deployment constraints of passive monitoring systems (i.e. active monitoring systems can be deployed at the edge of networks, when passive monitors must be deployed inside networks). The first generation of passive measurement equipment has been designed to collect packet headers at line speed on an on-demand basis. This generation of monitoring systems is best illustrated by the OC3MON [18], Sprint's IPMON [8] or NProbe [17] experience. Pandora [19] allows to specify monitoring components and this way provides greater flexibility in specifying the monitoring task. However, it differs from our approach in that it enforces a strict dependency among components and does not allow to dynamically load/unload some components.

Routers also embed monitoring software such as for example Cisco's Netflow [5]. Netflow collects flow level statistics on router line cards. Given the severe power and space constraints on routers, Netflow cannot store large amount of records but it exports all the information it capture to an external collector. This forces network operators to apply aggressive packet sampling (in the order of 0.1%) to reduce the data transfer rate from the routers.

Recently, the database community has approached the problem of Internet measurements. Several solutions have been proposed that deploy stream databases techniques. AT&T's Gigascope [6] is an example of a stream database that is dedicated to network monitoring. The system support a subset of SQL but it is proprietary and no measurement data is made publicly available. Other systems such as Telegraph [4] or PIER [11], IrisNet [10], Aurora [3] address the problem of continuous and distributed queries and as such are very relevant to CoMo.

## 8 Conclusion

We have presented the architecture of an open system for passive network monitoring. We have justified the design choices and indicated the three main open

issues that are crucial for the success of the monitoring infrastructure: query engine, resource management and security of the system.

There is a number of other issues that have not been addressed in this paper but are currently under investigation: (*i*) coordination of multiple CoMo system to respond to a query or balance the computation load; (*ii*) optimal placement of CoMo system as well as modules to guarantee visibility on the traffic even in presence of network failures or re-routing events; (*iii*) use of sampling for reducing the load on the system in a controlled fashion; (*iv*) how to port the current architecture to other hardware systems, such as routers or network processors.

## Acknowledgments

We thank Luigi Rizzo, Larry Huston, Pere Barlet, Euan Harris, Lukas Kencl, and Timothy Roscoe for their valuable feedback and comments on this work.

## References

1. A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of ACM Sigcomm*, Sept. 2004.
2. CAIDA: Cooperative Association for Internet Data Analysis. http://www.caida.org.
3. D. Carney et al. Monitoring streams - a new class of data management applications. In *Proceedings of VLDB*, 2002.
4. S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing of an uncertain world. In *Proceedings of CIDR*, 2003.
5. Cisco Systems. NetFlow services and applications. White Paper, 2000.
6. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of ACM Sigmod*, June 2003.
7. Endace. http://www.endace.com.
8. C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, R. Rockell, D. Moll, T. Seely, and C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 2003.
9. GEANT. http://www.dante.net.
10. P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4), Oct.
11. R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of VLDB*, 2003.
12. L. Huston, R. Sukthankar, R. Wickremesinghe, M. Stayanarayanan, G. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Usenix FAST*, Mar. 2004.
13. Internet2. http://www.internet2.org.
14. IP Flow Information eXport Working Group. Internet Engineering Task Force. http://www.ietf.org/html.charters/ipfix-charter.html.
15. X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the ACM Sensys*, Nov. 2003.
16. S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *USENIX Winter*, Jan. 1993.

17. A. Moore, J. Hall, E. Harris, C. Kreibech, and I. Pratt. Architecture of a network monitor. In *Proceedings of Passive and Active Measurement Workshop*, Apr. 2003.
18. NLANR: National Laboratory for Applied Network Research. http://www.nlanr.net.
19. S. Patarin and M. Makpangou. Pandora: A flexible network monitoring platform. In *Usenix*, 2000.
20. V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale internet measurement. *IEEE Communications*, 36(8), 1998.
21. RIPE Rèseaux IP Europèens. http://www.ripe.net.
22. tcpdump/libpcap. http://www.tcpdump.org.
23. University of Oregon Route Views Project. http://www.routeviews.org.
24. J. Xu, J. Fan, M. Ammar, and S. Moon. Prefix-preserving IP address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. Nov. 2002.

# TCP Anomalies: Identification And Analysis

Marco Mellia, Michela Meo and Luca Muscariello *

Dipartimento di Elettronica, Politecnico di Torino, Torino, Italy
{mellia,meo,muscariello}@mail.tlc.polito.it

*(Invited Paper)*

**Abstract.** Passive measurements have recently received large attention from the scientific community as a mean, not only for traffic characterization, but also to infer critical protocol behaviors and network working conditions. In this paper we focus on passive measurements of TCP traffic, main component of nowadays traffic. In particular, we propose a heuristic technique for the classification of the anomalies that may occur during the lifetime of a connection. Since TCP is a closed-loop protocol that infers network conditions and reacts accordingly by means of losses, the possibility of carefully distinguishing the causes of anomalies in TCP traffic is very appealing and may be instrumental to the deep understanding of TCP behavior in real environments and the protocol engineering.

**Keywords:** Internet Traffic Measurements, TCP Protocol, Computer Networks.

## 1 Introduction

In the last ten years, the interest in data collection, measurement and analysis to characterize Internet traffic behavior increased steadily. Indeed, by acknowledging the failure of traditional modeling paradigms, the research community focused on the analysis of the traffic characteristics with the twofold objective of understanding the dynamics of traffic and its impact on the network elements, and of finding simple, yet satisfactory, models, like the Erlang teletraffic theory in telephone networks, for designing and planning packet-switched data networks.

The task of measuring Internet traffic is particularly difficult for a number of reasons. First, traffic analysis is made very hard by the strong correlations both in space and time due to the closed-loop behavior of TCP, the TCP/IP client-server communication paradigm, and the fact that the highly variable quality provided to the end user influences the user behavior. Second, the complexity of the involved protocols, and of TCP in particular, is such that a number of phenomena can be studied only if a deep knowledge of the protocol details is

* This work was founded by the European Community "euroNGI" Network of Excellence.

exploited. And, finally, some of the traffic dynamics can be understood only if the forward and backward directions of flows are jointly analyzed.

From what mentioned above, it is clear that, since TCP plays a central role in the generation of Internet traffic, measurement tools should be equipped with modules for the analysis of TCP traffic, which do not neglect the occurrence of all those anomalies that strongly influence TCP behavior. In this context, the objective of this paper is to propose a new heuristic classification technique of anomalies that may occur during the lifetime of a TCP connection. In [1] a simple but efficient classification algorithm for out-of-sequence TCP segments is presented. The classification proposed in this paper is a modification and extension of that classification which allows the identification of a number of phenomena which were not previously considered, such as unneeded retransmissions and flow control mechanisms.

The proposed classification technique is applied to a set of real traces collected at our institution. The results show that a number of interesting phenomena can be observed through the proposed classification, such as the impact of the use of TCP SACK on the occurrence of unnecessary retransmissions, the relative small impact of the daily variation of the load on the occurrence of anomalies, the quite large amount of network reordering.

## 2 Methodology

The methodology adopted in this paper is a modification and extension of the one proposed for the first time in [1], in which authors proposed a simple but efficient classification algorithm for out-of-sequence packets in TCP connections and presented measurement results within the Sprint IP backbone. Similarly to what proposed by them, we adopt a passive measurement technique rather than using active probe traffic. The classification in [1] identifies out-of-sequence events due to i) necessary or unnecessary segment retransmissions by the TCP sender, ii) network duplicates, or iii) network reordering. Building over the same idea, we complete the classification by distinguishing other possible causes of out-of-sequence or duplicate packets. In particular, we also focus on the *cause* that triggered the segment retransmission by the sender. We analyze packet traces which record packets in both directions of a TCP connection: both data segments and ACKs are recorded.

Figure 1 sketches the evolution of a TCP connection: connection setup, data transfer, and connection tear down phases are highlighted. The measurement point (sniffer) is located in some point in the path between the Client and the Server. Both the IP layer and TCP layer overhead are observed by the sniffer, so that the TCP sender status can be tracked. A TCP connection starts when the first SYN from the client is observed, and terminates after either the tear-down

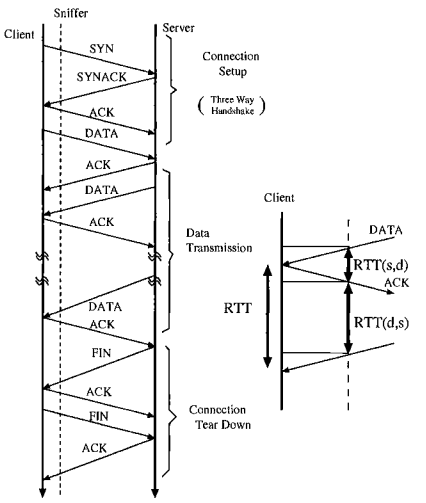



**Fig. 1.** Evolution of a TCP connection and the RTT estimation process.

sequence (the FIN/ACK or RST messages), or when no segment is observed for an amount of time larger then a given threshold [1].

By tracking the segment trace of a connection in both directions[2], the sniffer correlates the sequence number of TCP data segments to the ACK number of backward receiver acknowledgments and classifies the segments as,

- *In-sequence*: if the sender initial sequence number of the current segment corresponds to the expected one;
- *Duplicate*: if the data carried by the segment have already been observed before;
- *Out-of-sequence*: if the sender initial sequence number is not the expected one, and the data carried by the segment have never been observed before.

The last two classifications refer to an *anomaly* during the data transfer, that can have been caused by several reasons. We propose a fine heuristic classification of the anomalies. The classification has been implemented in [6, 7], and then used to analyze collected data. During the decision process that is followed to classify anomalies, several variables are used:

$RTT_{min}$ Minimum RTT: is the estimated minimum RTT observed since the flow started;

[1] Given that a tear-down sequence of a flow under analysis is never observed, a timer is needed to avoid memory starvation; the timer is set to 15 minutes, which is sufficiently large according to the findings in [2].

[2] In case only one direction of traffic is observed by the sniffer, the heuristic will not be applicable.

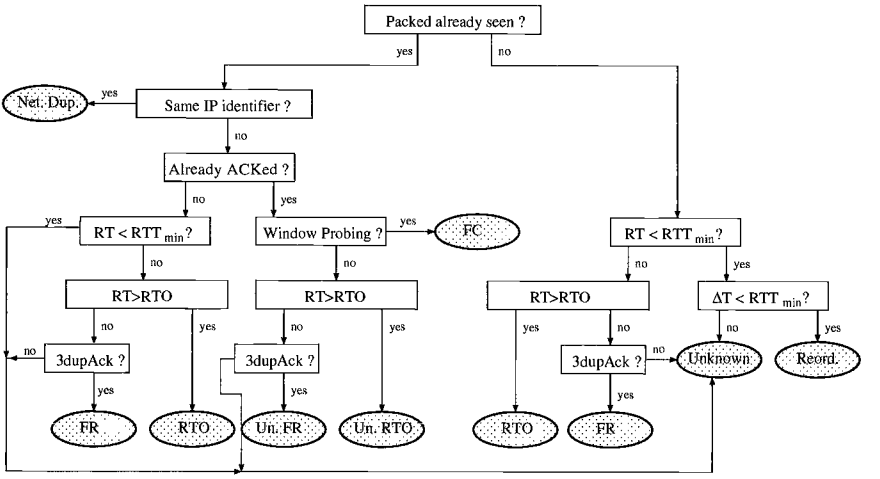



**Fig. 2.** Decision process of the classification of anomalous segments.

*RT* Recovery Time: is the difference between the time the current anomalous segment has been observed and the time the segment with the *largest* sequence number has been seen;

$\Delta T$ Inverted-packet gap: is the difference between the observation time of the current anomalous segment and the previously received one;

*RTO* Retransmission Timeout: is estimated sender retransmission timer value (in seconds) according to [3] as $RTO = max(1, E[RTT] + 4std(RTT))$;

*DupAck* Number of DupAcks: is the number of duplicate ACKs observed on the reverse path.

Following the flow diagram of Figure 2, we describe the classification heuristic. Given an anomalous segment, the process starts by checking if the segment has already been seen by comparing the TCP sequence number of the current segment with the one carried by segments observed so far. Thus, the segment can be classified as either duplicate or out-of-sequence. In the first case (left branch of the decision process), the IP identifier field of the current packet is compared with the same field of the original copy of the packet. If they are the same, then the packet is classified as **Network Duplicate** (Net. Dup.). Network duplicates may stem from malfunctioning apparatuses, routing loops, mis-configured networks (e.g., Ethernet LANs with diameter larger than the collision domain size), or, finally, by unnecessary retransmissions at the link layer (e.g., when a MAC layer ACK is lost in a Wireless LAN forcing the sender to retransmit the frame). Compared with the decision process adopted in [1], we classify as duplicate segments all those duplicate packets with the same IP identifier, regardless of the $\Delta T$ value. Indeed, there is no reason to exclude that a network duplicate may be observed at any time, and there is no relation between the *RTT* and the time a network can produce some duplicate packets.

When the IP identifiers are different, the TCP sender may have performed a retransmission. If all the bytes carried by the segment have already been acknowledged, then the receiver has already received the segment, and therefore this is an unneeded retransmission. The flow control mechanism adopted by TCP uses false unneeded retransmissions to perform *window probing*, i.e., to force the receiver to immediately send an ACK so as to probe if the receiver window $RWND$ (which was announced to be zero on a previous ACK) is now larger than zero. Therefore we classify as **Flow Control** (FC) retransmissions the retransmitted segments for which the following three conditions hold: i) the sequence number is equal to the expected sequence number decreased by one, ii) the segment size is of zero length, and iii) the last announced $RWND$ in the ACK flow was equal to zero. This is a new possible cause of unneeded retransmissions which was previously neglected in [1].

If the anomaly is not classified as flow control, then it must be an unnecessary retransmission, which could have been triggered because of either a Retransmission Timer (RTO) has fired, or the fast retransmit mechanism has been triggered, i.e., three or more duplicate ACKs have been received for the segment before the retransmitted one. We identify three situations: i) if the recovery time is larger than the retransmission timer ($RT > RTO$) the segment is classified as an **Unneeded Retransmission by RTO** (Un. RTO); ii) if 3 duplicate ACKs have been observed, the segment is classified as an **Unneeded Retransmission by Fast Retransmit** (Un. FR); iii) otherwise, if none of the previous conditions holds, we do not know how to classify this segment, and therefore we label it as **Unknown** (Unk.). Unneeded retransmissions may be due to a misbehaving sender, a wrong estimation of the $RTO$ at the sender, or, finally, to ACKs lost on the reverse path. However, distinguishing among these causes is impossible by means of passive measurements.

Let us now consider the case of segments that have already been seen but have not been ACKed yet. This is possibly the case of a retransmission following a packet loss. Given the recovery mechanism adopted by TCP, a retransmission can occur only after at least a RTT, since duplicate ACKs have to traverse the reverse path and trigger the Fast Retransmit mechanism. Therefore, if the recovery time is smaller than $RTT_{min}$, the anomalous segment can only be classified as **Unknown**[3]. Otherwise, it can either be a **Retransmission by Fast Retransmit** (FR) or **Retransmission by RTO** (RTO); the classification being based on the same criteria adopted previously for unneeded retransmissions. Retransmissions of already observed segments may be due to i) data segments lost on the path from the measurement point to the receiver, and to ii) ACK segments delayed or lost before the measurement point.

Consider now the right branch of the decision process, which refers to out-of-sequence anomalous segments. In this case, the classification criterion is simpler. Indeed, out-of-sequence segments can be due to either the retransmission of

[3] In [1] the authors use the average $RTT$. However, being each RTT possibly different than the average $RTT$ (and in particular smaller), we believe that using the average $RTT$ forces a larger amount of misclassification.

lost segments, or to network reordering. Again, since retransmissions can only occur if the recovery time $RT$ is larger than $RTT_{\min}$, by double checking the number of observed duplicate ACKs and by comparing the recovery time with the estimated $RTO$, we can distinguish retransmissions triggered by RTO, by FR or we can classify the segment as an unknown anomaly. On the contrary, if $RT$ is smaller than $RTT_{min}$, then a **Network Reordering** (Reord) is identified if the inverted-packet gap is smaller than $RTT_{min}$. Network reordering can be due to either load balancing on parallel paths, or to route changes, or to parallel switching architectures which do not ensure in-sequence delivery of packets [4].

### 2.1 Dealing with wrong estimates

The classification algorithm uses some thresholds whose values must be estimated from the packet trace itself, which may not be very accurate or even valid when classifying the anomalous event. Indeed, all the measurements related to the $RTT$ estimation are particularly critical, since they are used to determine the $RTT_{min}$ and the $RTO$ estimation. $RTT$ measurement is updated during the flow evolution according to the moving average estimator standardized in [3]. Given a new measurement $m$ of the RTT, we update the estimate of the average RTT by mean of a low pass filter $E[RTT]_{new} = (1-\alpha)E[RTT]_{old} + \alpha\ m$ where $\alpha$ $(0 < \alpha < 1)$ is equal to $1/8$.

Since the measurement point is not co-located at the transmitter, nor at the receiver, the measure of $RTT$ is not available. Therefore, in order to get an estimate of the $RTT$ values, we build over the original proposal of [1]. In particular, denote by $RTT(s,d)$ the *Half path RTT sample*, which represents the delay at the measurement point between an observed data segment flowing from the transmitter, or source, to the receiver, or destination, and the corresponding ACK on the reverse path, and denote by $RTT(d,s)$ the delay between the ACK and the following segment, as shown in Figure 1. From the measurement of $RTT(s,d)$ and $RTT(d,s)$ it is possible to derive an estimate of the total round trip time $RTT$,

$$RTT = RTT(s,d) + RTT(d,s)$$

The estimation of the average $RTT$ is not biased, given the linearity of the expectation operators. Therefore, it is possible to estimate the *average* $RTT$ by,

$$E[RTT] = E[RTT(s,d)] + E[RTT(d,s)]$$

Moreover, the standard deviation of the connection's RTT, $std(RTT)$ can be estimated following the same approach, given that $RTT(s,d)$ and $RTT(d,s)$ are independent measurements, which usually holds.

Finally, given $E[RTT]$ and $std(RTT)$, it is possible to estimate the sender retransmission timer as in [3]:

$$RTO = E[RTT] + 4std(RTT)$$

For what concerns the estimation of minimum RTT, $RTT_{min}$, we have

$$RTT_{\min} = \min(RTT(s,d)) + \min(RTT(d,s))$$

In general, this estimator gives a conservative estimation of the real minimum $RTT$, as $RTT_{\min} \leq \min(RTT)$ holds. This leads to a conservative classification algorithm, which increases the number of anomalies classified as unknown, rather than risking some misclassifications.

### 2.2 Handling Particular Cases

**No RTT Sample Classification.** There are some cases in which the $RTT$ measurement is not available, but an anomalous event is detected. This happens in particular at the startup of a TCP connection, as no valid $RTT$ samples may be available at the very beginning of the connection. Since most of TCP flows are very short [7], these events are quite frequent and cannot be neglected. Moreover, the choice of the initial values of $RTO$ and $RTT_{min}$ results to be critical, and inappropriate estimations of these variables may lead to wrong classifications. We adopt the following approach:

- if no valid RTT samples have been collected, the heuristic uses $RTO = 3s$ and $RTT_{min} = 5ms$ as default values
- the $RTO$ estimation is forced to assume values larger or equal to $1s$, according to [3]
- the $RTT_{min}$ estimation is forced to be larger than $1ms$

**Batch Classification.** Given that TCP can transmit more than one segment per RTT, it may happen that more than one anomalous segments are detected back-to-back. This occurs, for example, when the TCP sender adopts the SACK extension and retransmits more than one segment per RTT, or, when packets belonging to the same window on a path in which packets are reordered, arrive with "strange" patterns difficult to be identified. In such cases, the measurement of $RT$ and $\Delta T$ may be wrong and lead to incorrect classifications. We therefore implement a filter in the classification heuristic that correlates the classification of the current anomaly with the classification of the previous segment. In particular, if the current recovery time $RT$ is smaller than $E[RTT]$ (suggesting that the segment is belonging to the same window as the previous one) and the previous segment was not classified as in sequence, we then classify the current anomalous segment as the previous one.

For example, consider a simultaneous SACK retransmission of two segments triggered by a Fast Retransmit. The first retransmitted segment is correctly classified given that three duplicate ACKs have been observed on the reverse path, and the $RT$ is larger than $RTT_{min}$. However, the second retransmitted segment cannot be correctly classified, given that no duplicate ACK has ever been observed, and $RT < RTT$. By explicitly considering the classification of the first segment, it is possible to correctly identify this segment as a retransmission triggered by Fast Retransmit.

## 3 Measurement Results

Our measurements have been gathered from the external Internet edge link of our institution. Our campus network behaves like an Internet stub, because the access router is the sole gate to the external network. Our institution counts more than 7,000 hosts, whose great majority is constituted by clients. Some servers are regularly accessed from outside as well. We collected all packets flowing into the access link that connects the campus border router to the GARR network [5], the nation-wide ISP for research and educational centers. We measured for several months during several time periods and gathered the most interesting statistics related to the anomalous traffic. We present only a subset of results, and in particular:

- from the 6th to the 7th of February 2001. The bandwidth of the access link was 14 Mbit/s;
- from the 29th of April to the 5th of May 2004. The bandwidth of the access link was 28 Mbit/s.

### 3.1 Impact of $RTT_{min}$ and $RTO$

We first lead a set of measurements to double-check the impact of the choices described in Sec 2.1. In particular, we are interested in the impact of the measurement of $RTT_{min}$ and $RTO$. The first one is involved on the classification of the network reordering anomalies that may occur when identifying two out-of-sequence segments separated by a time gap smaller than $RTT_{min}$. Figure 3 (a) plots Cumulative Distribution Function (CDF) of the ratio between the inverted packet gap $\Delta T$ and the value of the $RTT_{min}$ considering only TCP anomalies classified as network reordering. Measurements referring to 2004 are reported, and similar results are obtained considering the 2001 dataset. The CDF clearly shows that $\Delta T$ is much smaller than the $RTT_{min}$. This suggests that the initial choice of $RTT_{min}$ is appropriate, and the conservative estimation of $RTT_{min}$ does not affect the classification.

Figure 3 (b), reports part of the CDF of the ratio between the actual Recovery Time $RT$ and the corresponding estimation of the $RTO$ when considering anomalous events classified as retransmissions by RTO. Also in this case we report results referring to the 2004 dataset. The CDF shows that $RT > RTO$ holds, which is a clear indication that the estimation of the RTO is not critical. Moreover, it can be noted that about 50% of the cases have a recovery time which is more than 5 times larger than the estimated RTO. This apparently counterintuitive result is due to the RTO back-off mechanism implemented in TCP but not considered by the heuristic which doubles the RTO value at every retransmission of the same segment. Not considering the back-off mechanism during the classification lead to a robust and conservative approach.

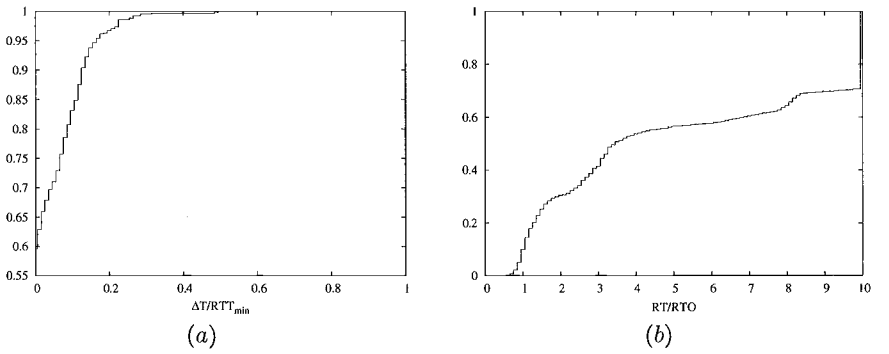



**Fig. 3.** (a) CDF of the ratio between the inverted packet gap $\Delta T$ and $RTT_{min}$. (b)CDF of the ratio between the recovery time $RT$ and the actual estimation of the RTO.

### 3.2 Aggregate Results

In the following we report results obtained by running the classification heuristic over the two datasets we selected and by measuring the average number of occurred anomalies during the whole time period. The objective is twofold. First, we quantify the different causes that generated anomalous segment delivery; second, we double check the heuristic classification. Indeed, it is impossible to test the validity of the classification algorithm, given that the real causes of the anomaly are unknown. We therefore run the classification over real traces, and try to underline some expected and intuitive results that confirm the validity of the heuristic design.

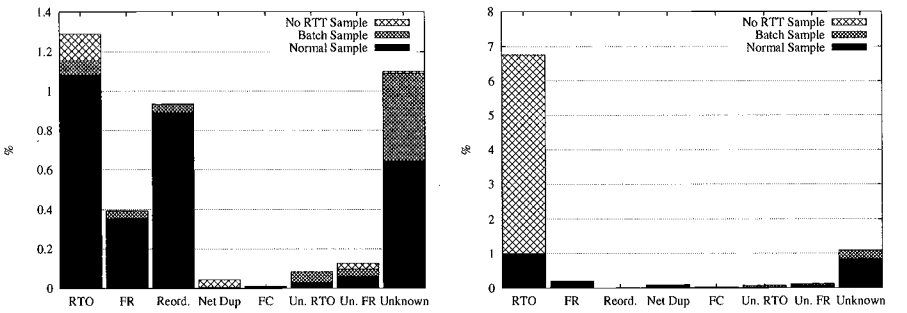



**Fig. 4.** Classification of anomalous events: incoming traffic on the left , outgoing traffic on the right.

Figure 4 reports the percentage of identified anomalous events during the 2004 period over the total amount of observed segments. Left plot refers to incoming traffic, i.e., traffic whose destination host is inside our campus LAN; right

plot reports measurements on outgoing traffic, i.e., traffic whose destination host is outside our campus LAN. Each bar in the plot explicitly underlines the impact of the batch classification and of the lack of RTT samples, as described in Section 2.1: solid black blocks report the anomalies classified by a normal classification, while dark pattern blocks report the impact of the batch classification, and, finally, light pattern blocks report the classification obtained when no RTT sample was available, i.e., at the very beginning of each flow.

Considering the incoming anomalies classification (left plot), we observe that there is a large dominance of retransmissions due to RTO expiration and reordering. Fast Retransmit occurs only for a very small portion of the total retransmissions. This is related to the characteristics of today data traffic, which is mainly composed of very small file transfers that cannot trigger Fast Retransmit. This effect is further stressed by the fact that our campus LAN traffic is mainly made of web browsing applications, whose (short) HTTP requests travel on the outgoing directions.

A small percentage of unnecessary retransmissions is also present. A rather large percentage of anomalies that could not be classified but unknown is collected. Inspecting further, we observed that for most of them the recovery time is smaller than the estimated RTO (therefore missing the retransmission by RTO classification), but larger than the $RTT_{min}$ (therefore missing the reordering classification) and the number of duplicate ACKs is smaller than 3 (therefore missing the retransmission by FR classification). We suspect that they may be due to either i) transmitters that trigger the Fast Retransmit with just 1 or 2 duplicate ACKs, or ii) servers with aggressive RTO estimation that triggers the retransmission earlier than the RTO estimation at the measurement point. Given the conservative approach that guided the classification heuristic, we prefer to classify them as unknown rather than misclassifying them.

For what concerns the impact of the batch classification and of the lack of a valid RTT sample, observe that the first is evenly distributed among all classification cases, while the latter one has a large impact on the identification of retransmissions by RTO. This is due to the lack of valid RTT samples at the very beginning of the TCP connection, when the sender can only detect packet losses by RTO.

When considering the outgoing anomalies (right plot), the heuristic correctly identifies the anomalies, and neither Network Reordering nor Duplicates are identified. Indeed, this is quite obvious given that our institution LAN is a switched Ethernet LAN. In this network, IP packets can be duplicated or reordered only in case of malfunctioning. This confirms the validity and robustness of the classification heuristic we developed.

Considering the average percentage of total anomalies identified,, we have that about 4% and 8% of incoming and outgoing traffic respectively is affected by an anomaly. We will see in the next section that the average values is not representative at all, given the non-stationaries of the anomalies.

Finally, in order to double check the validity of our heuristic, we split the flow into three different classes based on their segment length. *Short* flows (also

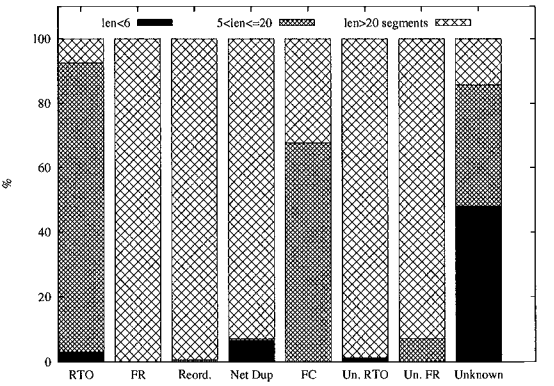



**Fig. 5.** Classification of anomalous events for different flow length; incoming traffic for 2004 measurements.

called mice in the literature) have payload size (in segments) no longer than 5 segments. *Long* flows (the so called elephants) have payload size (in segments) larger than 20 segments, and the *middle* length flows payload size is larger than 5 segments, but shorter or equal to 20 segments. Figure 5 reports the classification of the anomalous events split among the three different classes. Incoming segment classification is considered for the 2004 time period. Solid black refers to short flows, dark gray pattern refers to middle flows, and light gray pattern refers to long flows. For the sake of clarity, we omit the further batch or no rtt sample classification.

As expected, retransmissions due to RTO expiration are distributed among all flows, while retransmissions due to Fast Recovery are only triggered for long flows. This is intuitive, as already said, because of the limit in triggering Fast Retransmit by short flows. The majority of packet reordering affects long flows, for which the chance to suffer from a reordering is much larger. Neglecting the network duplicates, flow control and unnecessary retransmissions as they are very marginal, we observe that the anomalies classified as unknown are largely related to short flows. Indeed, this is an hint that the $E[RTT]$ estimation is affected by a larger error for short flows, while long flows have the chance to get a better estimation of the RTT and therefore to better classify the anomaly. This confirms the intuition that the unknown classification is related to possible different estimation of the RTO at the transmitter and at the measurement point.

Results relatively to outgoing flows and to the 2001 dataset are very similar and therefore not reported here.

### 3.3 Behavior in Time

We report in this section the results of the occurrence of anomalies in time. Again, we omit the sub-classification due to batch or no RTT samples for the

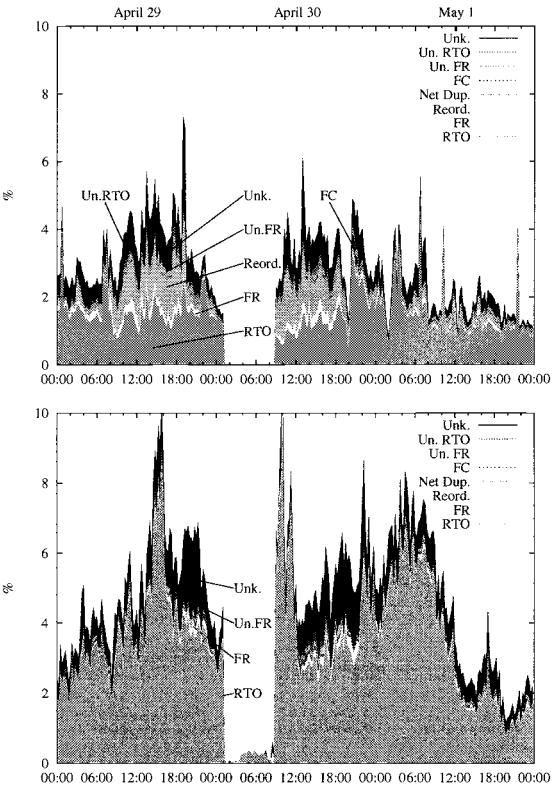



**Fig. 6.** Classification of anomalous events versus time: incoming traffic on the left, outgoing traffic on the right for 2004 measurements.

sake of clarity. Figures 6 and 7 depict the time evolution of the volume (in percentage, normalized on the total flowed traffic in each direction) of anomalous measured segments classified by the proposed heuristic. Measurements aggregate anomalies over an interval of time equal to 15 minutes. The detailed classification is outlined in colored slices whose size is proportional to the percentage of that particular event. Top plot refers to the incoming traffic; bottom plot reports measurements considering outgoing traffic. Figure 6 refers to the three days evolution of such traffic continuously monitored and classified during the 2004 period, while, similarly, Figure 7 refers to the two day long subset of measurements in 2001.

Apart from the network outage that is evident, the first unambiguous results is that TCP anomalies are highly non stationary over several time scales. There are some peaks of very significant magnitude that reach 10% during 2004 and 15% during 2001. Considering the incoming segments, Retransmissions by RTO, Network Reordering and Unknowns are the largest part of the anomalies, while TCP Flow control seldom kicks in, and negligible Unnecessary Retransmissions

are identified. Surprisingly, the typical night and day effect, which is commonly present on the total traffic volume (and is valid also in the considered link), is not anymore visible when considering TCP anomalies. Notice also that the last two measurement days are weekend-days. In particular, the Labour Day is celebrated on Sunday the 1st of May in Italy. Therefore the link load during that weekend was particular low. Nonetheless, the RTO fraction is almost equal to the one observed during busy hours of weekdays. Only the Network Reordering seems to disappear. This hints to a weak correlation between link load and TCP anomalies.

Considering the outgoing traffic (bottom plots of Figure 6 and 7), observe that the heuristic correctly identifies the anomalies as retransmission by RTO. Given that hosts in our campus LAN are mainly clients of TCP connections, the outgoing flow size is very short, and therefore in case of a packet loss, the only way to recover is to fire the RTO.

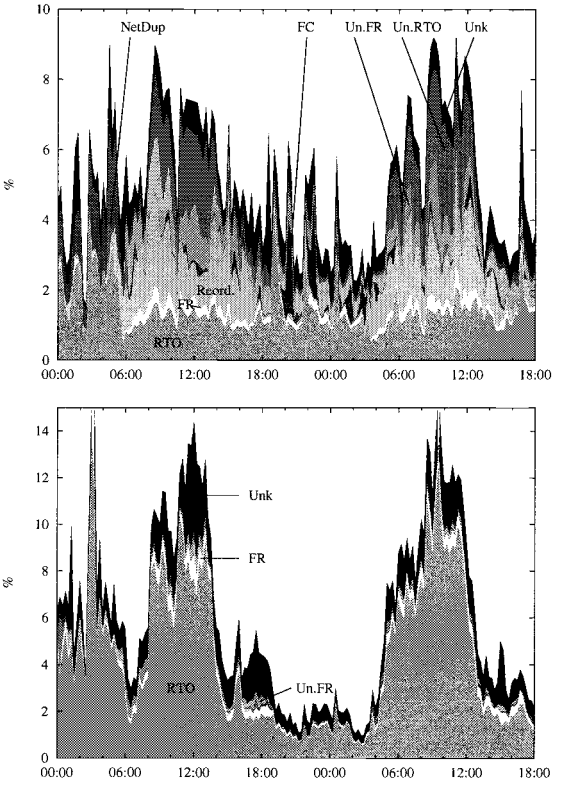



**Fig. 7.** Classification of anomalous events versus time: incoming traffic on the top, outgoing traffic on the bottom for 2001 measurements.

If we compare 2004 and 2001 incoming time plots, it can be noted that after three years the number of Unnecessary Retransmissions almost disappears. This fact is explained by the vast popularity of TCP SACK flows that were only the 21% of total flows during 2001 while it increased to 90% of total flows during 2004. At the same time, a reduction of the fraction of TCP anomalies is noticeable comparing 2001 and 2004 measurements. This could be related to the corresponding increase in the access link capacity, which doubled in 2004.

## 4 Conclusions

In this paper, we proposed a heuristic technique for the classification of TCP anomalies. The classification identifies seven possible causes of anomalies and extends previous techniques already proposed in the literature. We were also able to quantify the quality of our classification by studying the sensitivity of our approach in estimating the RTT which strongly influences the proposed methodology which gathers every flow state information from passive measurements and the quantitative analysis show its effectiveness in the identification procedure.

The technique was implemented and tested on the external Internet link of our institution and allowed the observation of a number of interesting phenomena such as the impact of the use of TCP SACK on the occurrence of unnecessary retransmissions, the relative small impact of the daily variation of the load on the occurrence of anomalies, the quite large amount of network reordering anomalies.

## References

1. S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, D. Towsley, "Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone", IEEE Infocom, San Francisco, March 2003.
2. G.Iannaccone, C.Diot, I.Graham, and N.McKeown, "Monitoring very high speed links," *ACM Internet Measurement Workshop, San Francisco*, November 2001.
3. V. Paxson, M. Allman, "Computing TCP's Retransmission Timer", *RFC 2988*, November 2000.
4. J.C.R.Bennett,C.C.Partridge, N.Shectman, "Packet reordering is not pathological network behavior" em IEEE/ACM Transactions on Networking, Vol. 7, N. 6, pp.789-798, December 1999.
5. "GARR Network" http://www.noc.garr.it/mrtg/RT.TO1.garr.net /polito.garr.net.html
6. "Tstat Web Page" http://tstat.tlc.polito.it/
7. M. Mellia, R. Lo Cigno, F. Neri,"Measuring IP and TCP behavior on edge nodes with Tstat", Computer Networks 47(3): 1-21, Jan. 2005.

# A High Performance IP Traffic Generation Tool Based On The Intel IXP2400 Network Processor

Raffaele Bolla, Roberto Bruschi, Marco Canini, and Matteo Repetto

Department of Communications, Computer and Systems Science,
DIST–University of Genova, Via Opera Pia 13, 16145 Genova, Italy
{raffaele.bolla, roberto.bruschi}@unige.it, marco@reti.dist.unige.it,
matteo.repetto@unige.it

**Abstract.** Traffic generation is essential in the processes of testing and developing new network elements, such as equipment, protocols and applications, regarding both the production and research area. Traditionally, two approaches have been followed for this purpose: the first is based on software applications that can be executed on inexpensive Personal Computers, while the second relies on dedicated hardware. Obviously, performance in the latter case (in terms of sustainable rates, precision in delays and jitters) outclasses that in the former, but also the costs usually grow of some order of magnitude. In this paper we describe a software IP traffic generator based on a hardware architecture specialized for packet processing (known as Network Processor), which we have developed and tested. Our approach is positioned between the two different philosophies listed above: it has a software (and then flexible) implementation running on a specific hardware only slightly more expensive than PCs.

**Keywords:** IP traffic generation, Network Processors, Network performance evaluation.

## 1 Introduction

The current Internet is characterized by a continuous and fast evolution, in terms of amount and kind of traffic, network equipment and protocols. Heterogeneous equipment, protocols and applications (also referred to as "network elements"), high transmission rates in most of Internet branches and the need of fast development (to quickly fulfil new services' requirements and to reduce time to market) are at present the most critical issues in Internet growth. In this context, every network element should be carefully tested before being used in real networks: an error in a device or a protocol could easily result in technical problems (e.g., packet losses, connection interruptions, degradation in performance) and it could lead to significative economic damage in production environments.

Many of the tests on a new network element require the ability to generate synthetic traffic off-line; such traffic should be sufficiently complex, fast (high-rates) and, in a word, realistic to cover a good deal of operating conditions. Thus,

the possibility to create realistic network traffic streams is very useful today, and it is essential to reduce development times and bugs in new network elements. Moreover, the availability of good synthetic traffic sources helps researchers in understanding network dynamics and in designing suitable modifications and improvements in current protocols and equipment.

The generation of traffic streams is a quite "simple" task in simulated environments; on the contrary, the generation of real traffic on a testbed setup involves some critical issues related to precision, scalability and costs. Until now, two antithetical approaches have been traditionally followed for network traffic generation. The first one is based on software applications that do not require specific hardware, but can be executed on general purpose machines, such as inexpensive Personal Computers (PC). The second one is based on the development of specialized hardware.

The characteristics and functionalities of software tools can be very sophisticated but, at the same time, they can maintain a high flexibility level and (often) offer an open source approach: in other words, they are modifiable and adaptable to the specific requirements of the experiments to carry out. The main drawback of this approach is the architecture of the PC, which limits the precision and the maximum reachable performance; this is a heavy limitation today, with Internet traffic that increases continuously and network equipment that must manage a huge amount of traffic (from Gigabits to Terabits per second).

Several techniques can be used in PC-based network traffic generation, depending on the final goal. Quite often, it is useful to generate only a single stream of IP packets, characterized by the inter-arrival time between two consecutive packets; this stream can be used to test network equipment such as routers and switches with standard performance analyses, as those defined in RFC 2544 [1] and 2889 [2]. Examples of applications oriented to this type of generation are *Iperf* [3] and *Netperf* [4]. These two tools generate IP packets by starting from a fixed structure and by varying some fields (e.g., the source and destination addresses, the TOS, etc.); more advanced applications are able to keep into account also the behaviour of the transport protocols, by representing the traffic generation as a set of file transfers over TCP or UDP (see, for example, *Harpoon* [5]). All these tools do not care about packet contents; however, when the element under test is an application (such as a Web server or a DNS), this is not acceptable. Thus, other tools have been designed with this goal, for example *WebPolygraph* [6]. One common drawback of the above cited software tools is that each of them can generate only a specific type of traffic stream, which is not representative of the heterogeneous traffic flowing on a generic Internet link. Software such as *Tcpreplay* and *Flowreplay* [7] log the traffic on some links and then generate identical streams, by possibly changing some parameters in the IP packets. Unfortunately, in this case, the traffic streams are always the same and the registration of long sequences of traffic requires large amounts of fast memory.

The second approach, namely the custom hardware devices, is the most popular solution in the industrial environment. Suitable architectures have been

designed (often by using also standard components) to minimize delays and jitters introduced in packet generation; here all the functionalities are implemented "near" the hardware, without the intermediation of a general purpose operating system (that may be present at a higher level to facilitate the configuration and the control of the instrument and the management of the statistical data). The development of such equipment is usually very expensive, based on proprietary solutions and carried out for professional and intensive use only. Thus, the final result is that the tools are configurable, but not modifiable or customizable, and very expensive. All these elements, together with the high final cost of each device, make this kind of tools not much attractive, especially for academic researchers or small labs. Examples of such products are Caldera Technologies *LANforge-FIRE* [8], CISCO IOS *NetFlow* [9], Anritsu *MD1231A IP/Ethernet Analyser* [10] and SmartBits *AX/400* [11].

Starting from these considerations, we have decided to build an open source traffic generator that could be situated in the middle between the existent approaches: it should be flexible and powerful enough to be useful in most of the practical situations, but less expensive than professional equipment. To realize all these objectives we need to work near the hardware, but we also need to implement all the functionalities in software, to reduce development costs and make customization effective. Among different technologies we have taken into account, we found our ideal solution in the Network Processors: these are devices conceived for fast packet processing, often with the high degree of flexibility needed for implementing the desired algorithms; moreover their cost is lower than custom hardware devices.

Several Network Processor architectures are available from different manufacturers [12]; we have found the Intel IXP2400 to be the best compromise between computation power and flexibility. Moreover, this chip is available on an evaluation board, the Radisys ENP-2611, which represents the cheapest way to access this kind of Network Processors. Until now, we know only another similar approach to the problem of traffic generation [13], but it implements a very simple structure, which is unable to generate realistic traffic. In this work we describe a structure for a high-speed IP/Ethernet traffic generator based on the Intel IXP2400 Network Processor that we have developed and tested. The requirements of this device are to be able to transmit more than 1 Gbps of traffic (to saturate all the Gigabit Ethernet lines of the ENP-2611) and to generate multiple traffic streams simultaneously, each of them with different statistical characteristics and header contents. Our aim is to model the more representative traffic classes in the Internet, such as real-time and Best Effort; moreover, we would like to have the chance to manage Quality of Service (by using the TOS field).

In the following, after briefly describing the IXP2400 architecture, we focus on two main issues. The first concerns the question if the Network Processor can be effectively utilized for traffic generation: the IXP2400 has been designed for packet processing, not explicitly for packet generation, and we were not sure that it can saturate the Gigabit interfaces, as we would like for a high-performance

tool. The second issue regards the design of an application framework that exploits the Network Processor architecture to build a flexible tool for traffic generation. We have named such framework PktGen and, as will described in detail later, it consists of several applications running on different processors of the IXP2400. PktGen is very simple to use, as it provides a comfortable user graphical web interface; moreover, it can also be modified without much effort, as it is mostly written in C. Preliminary tests have been carried out to demonstrate the correcteness and the performance of such tool.

The rest of the paper is organized as follows: Sect. 2 gives a brief overview of the Intel IXP2400, whereas the successive Sect. 3 explains how the Network Processor architecture is used to build a packet generator. Section 4 describes PktGen in details by analysing all its components, whereas in Sect. 5 we report some performance tests about PktGen and a comparison with UDPGen, a well-known software PC-based traffic generator. Finally, in Sect. 6 we give our conclusions and we report some ideas for future work.

## 2 The IXP2400 Architecture

Figure 1 shows the main components of the Intel IXP2400 architecture and the relationships among them. The IXP2400, as some other Network Processors, provides several processing units, different kinds of memory, a standard interface towards MAC components and some utility functions (e.g. hash and CRC calculation). The "intelligence" resides in the processing units, which can use the other peripherals though several internal buses.

For what concerns processing, the IXP is equipped with two kinds of microprocessors which play very different roles in the overall architecture: one XScale CPU and eight MicroEngines. The XScale is a generic RISC 32-bit processor, compatible with the ARM V5 architecture, but without the floating point unit. This processor is mainly used to control the overall system and to process network control packets (such as ICMP and routing messages). Due to its compatibility with the industry based standards (ARM), it is possible to use on it both the *Linux* and *VxWorks* [14] (derived from *Windows*) Operating Systems (OS); this is a great advantages as many existent applications and libraries can be compiled and used on this processor (greatly decreasing development time and increasing efficiency). For what concerns Linux, two different distributions are available at the moment: *Montavista* [15,16], based on a 2.4.18 kernel, and *Fedora* [17], with a more recent 2.6.9 kernel.

Accordingly to our aim of building an open-source tool, we have opted for Linux OS; in particular we have chosen the Montavista distribution, which was already available at the beginning of the project (Fedora has been released only recently). This choice enables us to develop software for this processor in standard C or C++ languages.

The other processing units of the IXP are the Microengines (ME). They are minimal RISC processors with a reduced set of instructions (about 50), optimized for packet processing; they provide logical and arithmetic operations

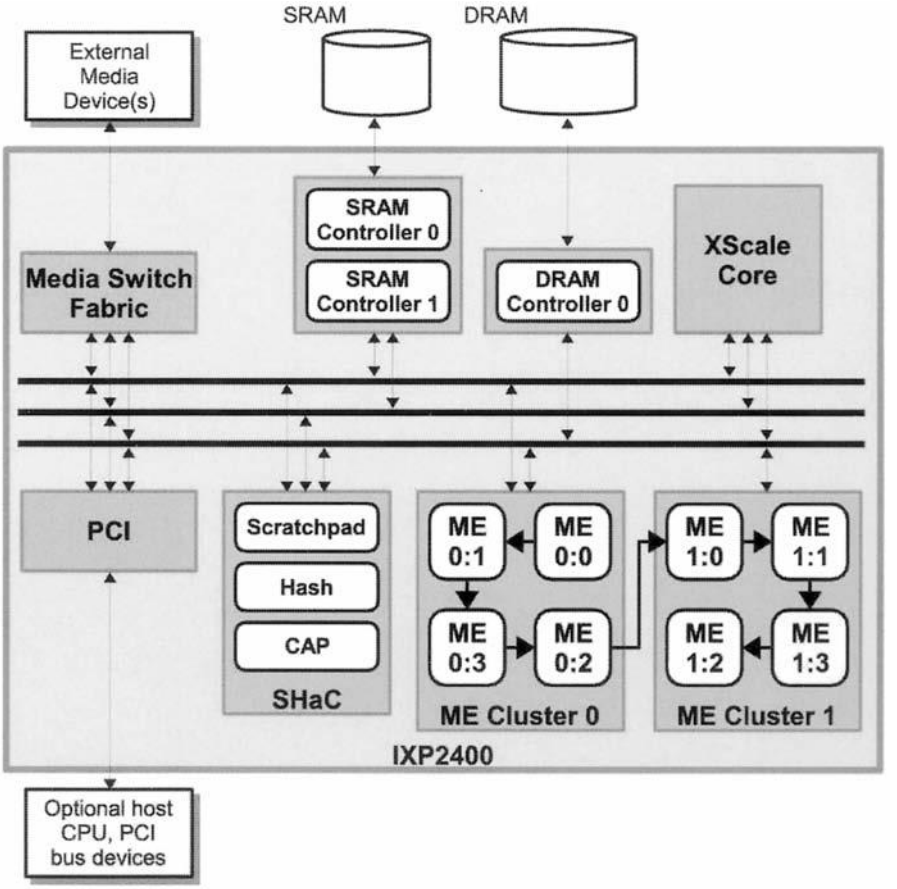



**Fig. 1.** The IXP architecture.

on *bits*, *bytes* and *dwords* but not the division nor the floating point functions. The MEs have access to all the shared units (SRAM, DRAM, MSF, etc.) and they are used along the *fast data path* for packet processing; they can be used in different ways (e.g., in parallel or sequentially) to create the framework that best fits the computational needs of network equipment.

No operating system is required on the ME, thus no software scheduler is available for multi-threaded programming. A simple round robin criterion is used to execute more threads (up to 8) on a ME; the programmer has the burden to write the code for each thread in a way that periodically releases the control of the ME to the other ones. Internally, each ME is equipped with specific registers for at most 8 hardware contexts (corresponding to the threads) and a shared low latency memory; moreover, some dedicated units are available for specific tasks (CRC computation and pseudo-random number generation).

ME programming can be made by assembler language or by microC; the latter has the same instruction set as C plus some non-standard extensions, and it makes the learning of the language itself and the programming of the MicroEngines simpler and faster.

The IXP2400 can use three types of memories: scratchpad, SRAM and DRAM. The scratchpad has the lowest latency time, it is integrated into the IXP2400 chip itself, but it is only 16 KB in size. The main feature of this memory is the support for 16 FIFO queues with automatic get and put operations, very useful to create transmission requests for a transmission driver.

The SRAM and the DRAM are not physically placed internally to the IXP-2400, but the NP chip provides only the controllers. The SRAM has a lower latency than the DRAM, but its controller supports up to 64 MB against 1 GB of the DRAM one. DRAM is usually used for storing the packets, while SRAM is reserved for the packet metadata, that are smaller but need to be accessed more frequently than packets themselves.

The last notable component is the *Media Switch Fabric* (MSF), that is an interface to transfer packets to/from the external MAC devices. Inside the IXP2400, the MSF can access directly the DRAM, resulting in high performance in storing/retrieving packets, while the external interface can be configured to operate in the UTOPIA, POS-PHY, SPI3 or CSIX modalities.

Finally, we can mention the hash unit, the PCI unit (used to provide a communication interface towards a standard external PCI bus) and the *Control and status register Access Proxy* (CAP), for the management of the registers used in the inter-process communication.

In our environment the IXP2400 is mounted on a Radisys ENP-2611 development board, that includes three optical Gigabit Ethernet ports (for fast-path data plane traffic), SRAM e DRAM sockets, a PCI connector (to access the IXP2400 SRAM and DRAM modules) and a further FastEthernet port for "direct" communications with the XScale processor.

A more detailed knowledge of the architectures of the IXP2400 and the Radisys development board is useless for the purpose of the paper. On the contrary, it is interesting to analyse how a standard packet processing phase takes place in the fast-path of the Network Processor.

Figure 2 shows a standard fast-path structure: packets are received from one Ethernet interface by a RX driver, passed to the Microengines (and eventually to the XScale) for the required processing and finally delivered to the TX driver for transmission over the physical line. The events that occur inside the Network Processor during this path crossing are the following:

1. A packet is received from the physical interface and delivered to the MSF.
2. The MSF buffers the packet and it notifies the RX driver, running on a MicroEngine.
3. The RX driver commands the MSF to transfer the buffered packet in the DRAM; then it creates a metadata structure for that packet, which is inserted in the SRAM. An identificator of the packet is put in a specific ring queue of the scratchpad, for successive processing.

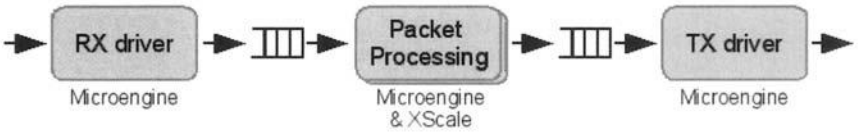



**Fig. 2.** A standard IXP2400 fast-path.

4. A Microengine thread gets the packet identificators (queued from the RX driver) and starts the packet processing phase. The operations that occur on the packet are application-specific and they may range from a simple routing between ports to more advanced features, such as *firewalling*, *natting*, etc; these operations can be carried out by a single thread or by multiple threads (each of them should perform a simple task), and sometime by the XScale. The latter is usually involved very rarely (e.g., it processes the routing update packet exchanged between routers or the control information directed to the Network Processor), as the presence of an operating system enables more complex operations, but results in slower execution times. During this phase the data are stored in the SRAM and both the packet data and metadata could be modified; finally the packet identifier is inserted in a ring queue for transmission (again in the scratchpad memory).
5. The transmission driver continuously checks the transmission ring queue, looking for new identifiers of the packets to be transmitted; each time it finds a new identifier, it gets the metadata from the SRAM and it instructs the MSF for transmission.
6. The MSF gets the packet from the DRAM and transfers it to the physical interface for the transmission.

## 3 Traffic Generation

In order to generate network traffic, the very basic tasks that we need to realize are the creation and transmission of the packets. The first issue to solve is the choice of where to locate the functions for creating and then transmitting the packets; in particular the two alternatives are the Xscale processor, which offers developers a well known environment and programming language, or the Microengines, faster for this task but less easily programmable.

In this context we have carried out several tests, in which the same algorithm (which transmits the same 60[1] bytes sized packet for a given number of times) has been implemented as:

[1] The actual packet size is 64 bytes, but the last 4 bytes are the CRC computed and appended by the PM3386 MAC device located on the Radisys board.

**Table 1.** Comparative results for different generation methods.

| *Generation method* | *Average pps*[3] *rate* |
|---|---|
| From XScale - kernel mode | 468 kpps |
| From XScale - user mode | 658 kpps |
| From Microengine | 842 kpps |

1. a kernel space application running on XScale, which has been developed by using the Resource Manager[2] library;
2. a user space application running on XScale, which uses the "*mmaped*" memory to access hardware features;
3. a microcode application running on a Microengine.

In the first and second case the algorithm has been implemented as C code and compiled by using the available GCC compiler with all the optimizations enabled. The effort required to develop a kernel module is greater then that required to build a user space application. However, given the availability of the Resource Manager library, the overall readability of code results enhanced with respect to the direct mapped memory usage in the user space application. On the other hand the Resource Manager library is specifically engineered for control plane tasks, thus it is not optimized to handle the transmission of packets efficiently.

For the third approach we have chosen to implement the algorithm in microC language, which is certainly easier than microengine assembly, while it preserves the same potential strength. In this case we were faced with a new programming model and payed extra time to get a base skill for it. Indeed, the results (Table 1) were not fulfilling the expectations and the attempts to tune the code did not give effective performance gains.

The main difference between generation from XScale (in both ways) and from Microengines is that in the first case we write a packet in DRAM for each transmission, while in the second case we always use the packets already present in the DRAM: actually, the high latency times of this kind of memory prevent the XScale code to keep up with higher packet generation rates.

For our purpose, it is more important to send similar packets at high rates rather than sending potentially completely different packets at low rates. Thus, we choose to have a set of packets (in the simplest case only one) always present in memory and to use Microengines to transmit each time one of the available packets. With this approach we can reach the maximum physical rate for a single port: 1.488 kpps, which for a packet size of 64 bytes is equivalent to 1 Gbps.

[2] Resource Manager is part of Intel IXA Portability Framework; although the IXA is not supported by the Radisys ENP-2611, in this case we succeeded in using part of it for our code.

[3] Packets per second

Obviously, transmitting the same packet can be of limited interest (especially for what regards header field contents, such as the source/destination addresses and the TOS); moreover, to store a great number of packets differing only for a few fields (or some combination of them) is not a clever solution. Thus, we introduced the concept of packet template to indicate a common structure of packet with a few fields that can be assigned dynamically at each transmission (according to some rules or some predefined sets of values), while the others have a fixed value.

We can make the concept of packet template clearer with an example. Suppose we want to generate a stream of UDP packets. Now suppose that we assign a fixed value to the source IP address and leave the destination IP address unspecified, since we have a list of possible destination addresses. Without the template mechanism we would create a duplicate of the same packet for each destination IP address; instead, with the template mechanism, only one packet is buffered and the list of possible destination IP addresses has to be passed to the packet generator software. The same software, at run time, will put one of those addresses in the packet's destination IP address field. In this way, a lot of memory has been saved and can be used for other packet templates.

The great benefit of this model is that we can reach the highest transmission rate while preserving all the flexibility required for a network traffic generator. The disadvantage is that all packet templates need to be available in memory before the generation can start.

This implies that the available physical memory represents an upper bound to the number of usable packet templates, but this is not a great issue, since the size of DRAM memory is large enough to store an amount of packet templates adequate for the characterization of many different traffic streams (the actual number of packets depends on their size).

## 4 The PktGen Framework

The IXP2400 architecture is well suited to be used to create a network traffic generator framework, including both a packet generator engine and a flexible and intuitive configuration interface. We have realized a software tool including three main components (see Fig. 3):

- the packet generator, an application written in microC and running on the MicroEngines, whose main task is the generation of packets with predefined statistical characteristics; using multiple instances of the application it is possible to simultaneously generate a set of traffic streams with homogeneous or heterogeneous characteristics;
- the PktGen core controller, an application running on the XScale, with the goal to initialize and control the packet generators on the MEs, including the packet templates to be used;
- a graphical interface, HTML based and accessible by a standard web browser, used to easily configure the packet generator parameters.

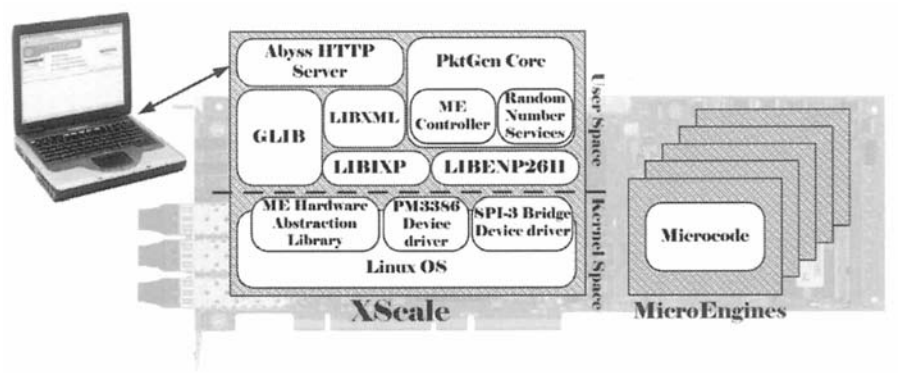



**Fig. 3.** Software architecture of PktGen.

At present PktGen is able to generate two kinds of traffic, which we consider very significant in testing IP-devices; they can be identified as *Constant Bit Rate* (CBR) and *Best Effort* (BE) or *bursty* streams.

For the BE streams we use a bursty model in which packets are generated in random-sized bursts with random inter-arrival times. The statistics of the two random variables can be modified quite simply, by passing the related probability density functions to PktGen; in our tests, we have used an exponential density function for both the inter-arrival times and the burst sizes. We think to introduce more kinds of traffic in the future; the structure of PktGen is flexible enough to perform this task in a simple and quick manner.

### The Packet Generator

The packet generator is a microC code compiled to run on a single MicroEngine in 4-threads mode. This means that actually there are 4 instances of packet generators concurrently executing on the same MicroEngine. Multiple instances of packet generators can run on more MicroEngines without conflicting with one another; in our configuration 5 MicroEngines are reserved for the packet generation task, thus we are free to simultaneously use up to 20 packet stream generators. Indeed, up to 7 MicroEngines can be used for packet generators; one ME must be reserved to the TX driver.

This PktGen component has been engineered to efficiently handle the generation of packets for both CBR and BE traffic models. Independently of the traffic model, the packet generator provides a setup and an operational interface controlled by the core component. The references to packet templates are forwarded to the packet generator through the setup interface, while the operational interface is used to control the generator state (running, stopped).

The two main tasks of the packet generators are the insertion of the variable fields in packet templates and the generation of random variables for statistical traffic characterization (inter-arrival times and burst sizes). At present, we have

chosen to work with 4 variable fields in the packet template: Source and Destination IP addresses, TOS value and Total Length. Indeed, there is a fifth dynamic field, the Header Checksum, but it is defined indirectly by the values of the other fields. Moreover, each of the four instances running on the same MicroEngine is able to store up to 160 packet template references: thus, we can obtain a great number of packet patterns that can differ, for example, for payload contents (protocol type and data).

To enable statistical traffic characterization, a hardware uniform pseudo-random number generator is used in conjunction with a set of off-line samples for an arbitrary probability distribution function (that are evaluated by the core controller at setup time, as explained in the following) to dynamically generate values for a given probability density function.

### The Core Controller

The core controller component of PktGen running on the XScale processor is built in C language. As shown in Fig. 3, this component is located on top of the software stack that starts from the Linux kernel. The two major sub-components are the ME Controller and Random Number Services. The former allows the PktGen Core Controller to start, stop and setup packet generators, by encapsulating all the hardware details (features coming from kernel modules as ME Hardware Abstraction Library, PM3386 and SPI-3 device drivers) and by interfacing with the user space libraries provided with the IXP2400 (*libixp* and *libenp2611*, see Fig. 3). The latter is dedicated to handle all the mathematics involved in computing the samples for a given probability density function that are passed to packet generators and used at run time to give a statistical characterization to each traffic stream.

The flexibility of changing the probability density function is one of the main features of PktGen; however, the lack of a floating point unit and the scarceness of memory make the random number generation one of the most critical issues of the entire framework. For this reason a few more words on this topic are needed.

By using the graphical interface the user can specify the analytical formula of any desired density function; the core controller can compute the distribution function (by means of the *libmatheval* [18] and *gsl*[4] [19] libraries) used to find a set of values with the inversion method [20]. Such values are first converted into an integer representation (despite the lack of the floating point unit the XScale can work with this kind of numbers by means of software libraries, but the MEs cannot) and then passed to the packet generator that randomly picks up one of them with a uniform distribution.

The set of values representative of the desired probability density function must be stored in the SRAM memory, because the latency of the DRAM is too high. Unfortunately, the SRAM on the Radisys board is only 8 MB; thus we cannot store a great number of probability samples; the user can choose to use 256 or 64K samples (corresponding to one or two byte per sample).

[4] Gnu Scientific Library

**Fig. 4.** Testbed 1: Measuring the maximum performance of PktGen.

Clearly, the integer conversion and the use of a limited number of samples (up to 64K) introduce a precision loss in the final traffic statistical characterization; we are still investigating the effects on the traffic generation and looking for techniques to minimize the errors introduced.

### The Graphical Interface

A simple, yet powerful, WEB-based interface is provided to facilitate the use of PktGen. This interface allows the user to create a different configuration for each desired traffic profile, to save the configuration for later reuse and to run simulations.

To ensure fast code development and maintenability we have used the Abyss HTTP server to provide the graphical interface, while to conform to standard file formats used by many existent tools we have adopted an XML coding of the configuration files.

## 5 Results

The main result of our activity is PktGen itself: it is a proof that our initial objectives were feasible and we succeeded in achieving them. As a matter of fact, we were able to fully saturate all the three Gigabit interfaces of the Radisys board.

Nonetheless, we are interested in a thorough evaluation of the performance of our tool, expecially for what concerns precision in generation (conformance of packet flows to the statistical description); most of these tests are planned for the immediate future, but some of them have been already carried out and can give an idea of the potentialities of the framework.

The main difficulties we have met concern our lack of a measurement powerful enough testing our generator. The PC architecture cannot sustain the packet rate from PktGen and thus no software tools can be used as meters; indeed, we would need hardware devices that are currently out of the project's budget.

Thus, we have carried on only a few simple measurements on PktGen performance, and to do this we have written a simple packet meter in microC to run it on one MicroEngine. This tool can measure the mean packet rate for any kind of traffic and the jitter for CBR traffic only.

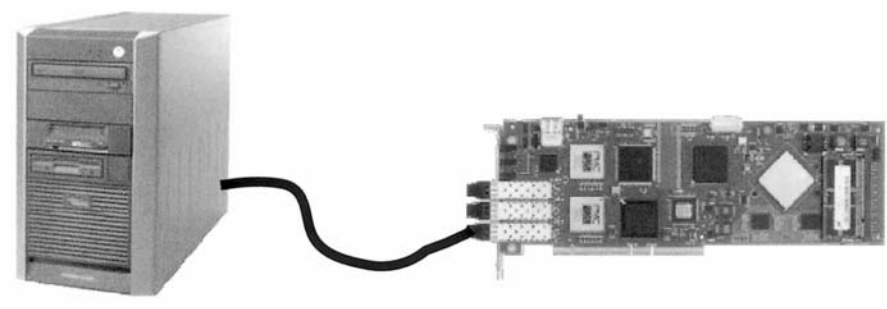
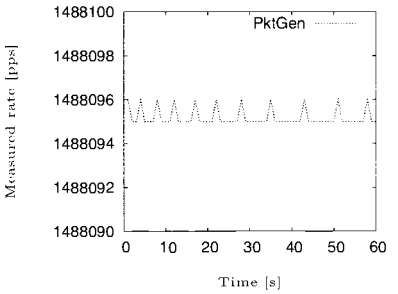
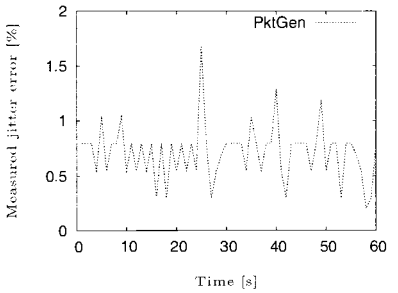


**Fig. 5.** Testbed 2: Comparison with generation from UDPGen.




(a) Measured generation rate over 60 s.



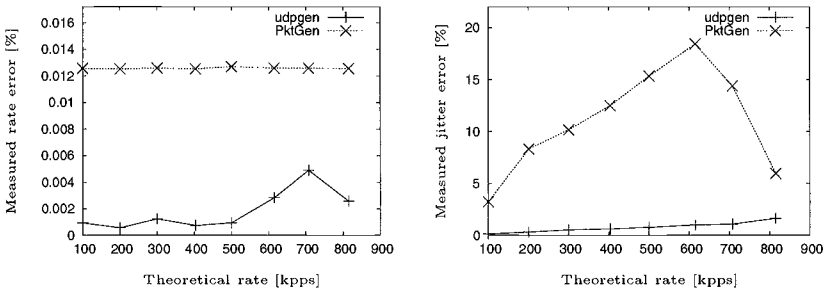

(b) Delay variations (jitters) at peak rate.

**Fig. 6.** Maximum performance of PktGen.

Our measurement tests have been devoted to two main issues: the first was to evaluate the PktGen maximum performance (see Fig. 4), and the second was to compare PktGen with a software tool running on a high-end PC (Fig. 5).

The PktGen maximum performance has been evaluated by means of the testbed shown in Fig. 4, where PktGen runs on one Network Processor and the other is used as the meter. In the worst case of minimum packet size (64 bytes), as reported in Fig. 6, we were able to generate a maximum Constant Bit Rate traffic of 1488.096 kpps, corresponding to 1 Gbps, with a high level of precision: the maximum measured jitter is below 2% of packet delay interarrival time in the CBR stream.

In the second testbed, we compared PktGen with a well-known software tool, namely UDPGen. The latter is used to transmit UDP packets and thus is quite similar to our application, which, however, works at the IP layer. We can see from Fig. 7 how PktGen overcomes UDPGen in terms of precision for what concerns both mean rate and jitter. Again, tests have been carried out for CBR traffic in the worst case of 64 byte packet size. Finally, it is worth noting that




(a) Error of measured rate vs desired rate.

(b) Jitter at different desired rates.

**Fig. 7.** Comparison between PktGen (running on the Network Processor) and UDPGen (running on a high-end PC) with Constant Bit Rate traffic.

this analysis is limited to a maximum mean rate of 814 kpps, because this is the peak rate obtainable with UDPGen (whereas PktGen can reach about 1488 kpps).

## 6 Conclusions and Future Work

We have shown that high-performance traffic generation with a Network Processor (in particular with the Intel IXP2400) is possible.

The main result of our activity, namely PktGen, is an open source framework written mostly in C and is easily customizable by everyone; also the microcode used in the MicroEngines is quite intuitive and simple to learn. PktGen is also able to fully saturate the three Gigabit interfaces of the development board (the ENP-2611), reaching a maximum aggregate packet generation rate of about 4464 kpps (3 Gbps).

We can therefore conclude that our aim is completely achieved: we have demonstrated how it is possible to build an open source, customizable, high-speed and precise packet generator without specific hardware; the cost is less expensive than that of a professional device (the Radisys ENP-2611 costs about $ 5000).

We consider this only a first important step in this field: in fact, we think that more work has to be done in this direction.

First of all we have planned to extend the PktGen functionalities to include also a packet meter, able to collect detailed and precise information on network traffic streams (mean rate, jitter, packet classification, etc.); this is necessary to have a counterpart to the generator, with the same noteworthy characteristics (open source, low cost, high performance and precision).

The second important goal is to port PktGen to a more performing architecture, for example to the IXP 2800, in order to realize a more and more powerful traffic generator.

Other minor evolutions are foreseen, such as the introduction of additional traffic shapes in PktGen, the comparison with professional devices and its utilization in our research activities on Quality of Service, high-speed networks, and Open Router architecture [21].

## 7 Acknoledgements

We would like to thanks Intel for its support to our work. Intel gave us the two Radisys ENP-2611 development board with IXP2400 Network Processor that we have used for the development of PktGen and have funded our research in the last year.

## References

1. Bradner, S., McQuaid, J.: Benchmarking methodology for network interconnect devices. RFC 2544, IETF (1999) Available online, URL: http://www.ietf.org/rfc/rfc2544.txt.
2. Mandeville, R., Perser, J.: Benchmarking methodology for LAN switching devices. RFC 2889, IETF (2000) Available online, URL: http://www.ietf.org/rfc/rfc2889.txt.
3. Tirumala, A., Qin, F., Dugan, J., Ferguson, J., Gibbs, K.: Iperf. Available online, URL: http://dast.nlanr.net/Projects/Iperf/ (2005)
4. Jones, R., Choy, K., et al.: Netperf. Available online, URL: http://www.netperf.org/ (2005)
5. Sommers, J., Barford, P.: Self-configuring network traffic generation. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement 2004, Portland, OR - USA, IMT 04 (2004)
6. Rousskov, A., Wessels, D.: Web Polygraph. Available online, URL: http://www.web-polygraph.org/ (2004)
7. Turner, A.: Tcpreplay. Available online, URL: http://tcpreplay.sourceforge.net/ (2005)
8. Caldera Technologies, C.: LANforge-FIRE. Available online, URL: http://www.candelatech.com/ (2005)
9. Cisco Systems, C.: IOS netflow feature. Available online, URL: http://www.cisco.com/en/US/tech/tk812/tsd_technology_support_protocol_-home.html (2005)
10. Anritsu, C.: IP/Ethernet analyser, Model: MD1231A. Available online, URL: http://www.eu.anritsu.com/products/default.php?p=97&model=MD1231A (2005)
11. Smartbits: AX/400. Available online, URL: http://www.netcomsystems.com/ (2005)
12. Comer, D.E.: Network Systems Design using Network Processors - Agere version. Pearson Prentice Hall, Upper Saddle River, New Jersey - USA (2005)
13. University of Kentucky, L.f.A.N.: IXPKTGEN project. Available online. URL: http://protocols.netlab.uky.edu/ esp/pktgen/ (2004)

14. Wind River, C.: Wind River Operating Systems. Available online, URL: http://www.windriver.com/products/device_technologies/os/ (2005)
15. Montavista: Montavista Linux Preview kit. Available online, URL: http://www.mvista.com/previewkit/index.html (2004)
16. Montavista: Montavista Linux Professional Edition. Available online, URL: http://www.mvista.com/products/pro/ (2004)
17. Buytenhek, L.: Port of Fedora for XScale processor". Available online, URL: http://skrybele.wantstofly.org/ (2005)
18. GNU: Libmatheval library. Available online, URL: http://www.gnu.org/software/libmatheval/ (2005)
19. GNU: Gsl library. Available online, URL: http://www.gnu.org/software/gsl/ (2005)
20. L'Ecuyer, P. In: Random Number Generation. Handbook of Computational Statistics. Springer-Verlag (2004) pp. 35–70
21. Bolla, R., Bruschi, R.: A high-end linux based open router for IP QoS networks: tuning and performance analysis with internal (profiling) and external measurement tools of the packet forwarding capabilities. In: Proc. of the 3rd International Workshop on Internet Performance, Simulation, Monitoring and Measurements (IPS MoMe 2005), Warsaw - Poland, Institute of Telecommunications, Warsaw University of Technology (2005)

# IP Forwarding Performance Analysis In The Presence Of Control Plane Functionalities In A PC-Based Open Router

Raffaele Bolla, Roberto Bruschi

Department of Communications, Computer and Systems Science (DIST)
University of Genoa
Via Opera Pia 13, I-16145 Genova, Italy
Email: raffaele.bolla, roberto.bruschi@dist.unige.it

**Abstract.** Nowadays, networking equipment is realized by using decentralized architectures that often include special-purpose hardware elements. The latter considerably improve the performance on one hand, while on the other they limit the level of flexibility. Indeed, it is very difficult both to have access to details about internal operations and to perform any kind of interventions more complex than a configuration of parameters. Sometimes, the "experimental" nature of the Internet and its diffusion in many contexts suggest a different approach. This type of need is more evident inside the scientific community, which often encounters many difficulties in realizing experiments. Recent technological advances give a good chance to do something really effective in the field of open Internet equipment, also called Open Routers (ORs). The main target approached in this paper is to extend the evaluation of the OR forwarding performance proposed in [1], by analyzing the influence of the control plane functionalities.

**Keywords**. Open Router, Linux kernel, IP forwarding performance

## 1 Introduction

During the past 20 years, Internet equipment has radically changed many times, to meet the increasing quantity and the complexity of new functionalities [2]. Nowadays, current high-end IP nodes belong to the 3rd generation, and are designed to reach very high performance levels, by effectively supporting high forwarding speeds (e.g., 10 Gbps). With this purpose, these devices are usually characterized by decentralized architectures that often include custom hardware components, like ASIC or FPGA circuits. On one hand, the dedicated hardware elements improve the performance considerably, while, on the other hand, they limit the level of flexibility. Moreover, with respects to this commercial equipment, it is very difficult both to have access to internal details and to perform any kind of interventions that would require more complex operations than those involved by a configuration of parameters: in this case, the "closure" to external modifications is a clear attempt to protect the industrial investment.

In many contexts the "experimental" nature of the Internet and its diffusion suggest a different approach. This type of need is more evident within the scientific community, which often finds many difficulties in realizing experiments, test-beds and trials for the evaluation of new functionalities, protocols and control mechanisms. But also the market frequently asks for a more open and flexible approach, like that suggested by the Open Source philosophy for software. This is especially true in those situations where the network functions must be inserted in products, whose main aim is not limited to realizing basic network operations.
As outlined in [1] and in [3] (that report some of the most interesting results obtained in the EURO project [4]), the recent technology advances give a good chance to do something really effective in the field of open Internet devices, sometimes called Open Routers (ORs) or Software Routers. This possibility comes, for what concerns the software, from the Open Source Operating Systems (OSs), like Linux and FreeBSD (which have sophisticated and complete networking capabilities), and for what concerns the hardware from the COTS/PC components (whose performance is always increasing, while their costs are decreasing). The attractiveness of the OR solution can be summarized in multi-vendor availability, low-cost and continuous update/evolution of the basic parts, as assumed by Moore's law.
The main target approached in this paper is to extend the evaluation of the OR forwarding performance proposed in [1], by analyzing the influence of the control plane functionalities. In fact, we have to outline that in a PC-based OR, unlike in much industrial dedicated equipment, the data and the control plane have to share the computational resources of the CPU(s) in the system. Thus, our main objective is to study and to analyze, both with external (throughput) and internal (profiling) measurement tools, if and how the presence of heavy control operations may affect the performance level of the IP forwarding process.
The paper is organized as in the following. The next Section describes the architecture of the open node for what concerns both hardware and software structure. Section III reports the parameter tuning operations realized on the OR to obtain the maximum performance. Section IV describes the external and internal performance evaluation tools used in the tests, while Sections V reports the numerical results of all the most relevant experiments. The conclusions and future activities are in Section VI.

## 2 Open Router Architecture

To define the OR reference architecture, we have established some main criteria and we have used them to a priori select a set of basic elements. The objective has been to obtain a high–end node base structure, able to support top performance with respect to IP packet forwarding and control plane elaborations. The decision process, the criteria and the final selection results are described in some detail in the following, separately for hardware and software elements.

### 2.a Hardware Architecture

The PC architecture is a general-purpose one and it is not specifically optimized for network operations. This means that, in principle, it cannot reach the same

performance level of custom high-end network equipment, which generally uses dedicated HW elements to handle and to parallelize the most critical operations. This characteristic has more impact on the data plane performance, where custom devices usually utilize dedicated ASIC, FPGA, Network Processor and specific internal bus, to provide a high level of parallelism in the packet processing and exchange. On the other hand, COTS hardware can guarantee two very important features as the cheapness and the fast and continuous evolution of many of its components. Moreover, the performance gap might be not so large and anyway more than justified by the cost difference.

The PC internal data path uses a centralized I/O structure composed by: the I/O bus, the memory channel (both used by DMA to transfer data from network interfaces to RAM and vice versa) and the Front Side Bus (FSB) (used by the CPU with the memory channel to access to the RAM during the packet elaboration). It is evident that the bandwidth of these busses and the PC computational capacity are the two most critical hardware elements involved in the determination of the maximum performance in terms of both peak passing bandwidth (in Mbps) and maximum number of forwarded packets per second.

For example, a full duplex Gigabit Ethernet Interface makes immediately inadequate the traditional PCI bus (32 bits with a frequency of 33 MHz). With these high-speed interfaces we need at least one of the two newest evolutions of this standard, namely PCI-X and PCI-Express, which can assure the needed overall bandwidth to handle the traffic of several Gigabit Ethernet interfaces. So, the selection criteria have been very fast internal busses and a dual CPU system with high integer computational power.

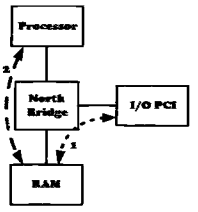



**Fig. 1**. Scheme of the packet path in a PC hardware architecture.

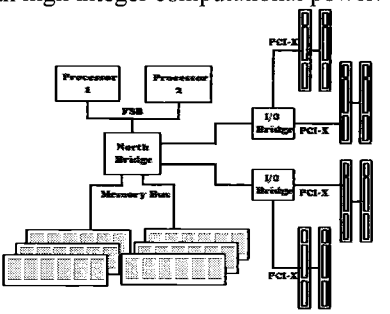



**Fig. 2**. Scheme of the Supermicro X5DL8-GG main board.

With this goal, we have chosen the Supermicro X5DL8-GG main board, mounting a ServerWorks GC-LE chipset, whose structure is shown in Fig. 2. This chipset can support a dual-Xeon system with a dual memory channel and a PCI-X bus at 133 MHz, with 64 parallel bits. The Xeon processor has the Pentium 4 core that allows multiprocessor configurations and Hyper-Threading (HT) capability enabled. It derives from the Intel NetBurst architecture and it is one of the most suitable 32 bit processors for high-end server architectures, thanks to its large L2/L3 cache size. In our tests we used a Xeon version with 2.4 GHz clock and 512 KB sized cache. The memory bandwidth, supported by this chipset, matches the system bus speed, but the most important point is that the memory is 2-way interleaved, which assures high performance and low average access latencies. The bus that connects the North Bridge to the PCI bridges, namely IMB, has more bandwidth (more than 25 Gbps)

than the maximum combined bandwidth of the two PCI-X busses (16 Gbps) on each I/O bridge.
Network interfaces are another critical element in the system, as they can heavily condition the PC Router performance. As reported in [5], the network adapters on the market have different levels of maximum performance and a different configurability. In this respect, we have selected the Intel PRO 1000 XT Server network interfaces, which are equipped with a PCI-X controller, supporting the 133 MHz frequency and also a wide configuration range for many parameters, like, for example, transmission and receive buffer lengths, maximum interrupt rates and other important features [6].

### 2.b Software Architecture

The software architecture of an OR has to provide many different functionalities: from the ones directly involved in the packet forwarding process to the ones needed for control functionalities, dynamic configuration and monitoring. In particular, we have chosen to study and to analyze a Linux based OR framework, as it is one of the open source OSs that have a large and sophisticated kernel-integrated network support, it is equipped with numerous GNU software applications. and it has been selected in the last years as framework for a large part of networking research projects.
Moreover, we have also decided not to take into account some software architectures, extremely customized to the network usage (e.g., Click [7], [8]): even if they often provide higher performance with respect to the "standard" frameworks, they often result compatible with few hardware components and standard software tools. For example, Click provides (by replacing the standard network kernel) a very fast forwarding mechanism [9], which is realized by polling the network interfaces, but it needs particular drivers (not yet available for many network adapters), and it is not fully compatible with many well-known network control applications (e.g., Zebra/Quagga).
For what concerns the Linux OR architecture, as outlined in [1] and in [3], while all the forwarding functions are realized inside the Linux kernel, the large part of the control and monitoring operations is running as daemons/applications in user mode. Thus, we have to outline that, unlike most of the commercial network equipment, the forwarding functionalities and the control ones have to share the CPUs in the system. In fact, especially the high-end commercial network equipment provides separated computational resources for these processes: the forwarding is managed by a switching fabric, which is a switching matrix, often realized with ad hoc hardware elements (ASICs, FPGAs and Network Processors), while all the control functionalities are executed by one ore more separated processors.
Let us now go into the details of the Linux OR architecture: as shown in Fig. 3 and previously sketched, the control plane functionalities run in the user space, while the forwarding process is entirely realized inside the kernel.
In particular, the networking code of the kernel is composed by a chain of three main modules, namely receiving API (RxAPI), IP Processing, transmission API (TxAPI), which manage respectively the reception of packets from the network interfaces, their IP layer elaboration and their transmission to the network devices. Moreover, the first

of these modules, the RxAPI, has experienced many structural and refining developments, as it is a very critical element for performance optimization.
The NAPI (New API) [10] is the latest version of RxAPI architecture, it is available from the 2.4.22 kernel, and it has been explicitly created to increase the system scalability, as it can handle network interface requests with an interrupt moderation mechanism that allows to adaptively switch from a classical interrupt management of the network interfaces to a polling one.

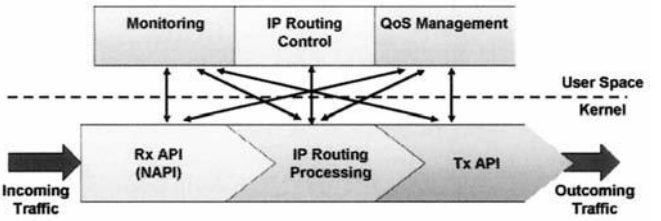



**Fig. 3**. Software architecture of a Linux-based OR.

Another important element of the kernel integrated forwarding architecture is the memory management. All the kernel code is structured with a zero-copy statement [10]: to avoid unnecessary and onerous memory transfer operations of packets, they are left to reside in the memory locations used by the DMA-engines of ingress network interfaces, and each operation on the packets is performed by using a sort of pointer to the packet and to its key fields, called descriptor or *sk_buff*. These descriptors are effectively composed by pointers to the different fields of the headers contained in the associated packets.
Moreover, the *sk_buff* data structure allows to realize all the buffers used by the networking layers, as it provides the general buffering and flow control facilities needed by network protocols. A descriptor is immediately allocated and associated to each received packet in the sub-IP layer, it is used in every subsequent networking operation, and finally de-allocated when the network interface signals the successful transmission.
The network code of the newest versions of the Linux kernel allows managing the forwarding process with Symmetric MultiProcessing (SMP) configurations. In fact, starting from version 2.0 on, the networking architecture of the Linux kernel has been developed by using a thread-based structure. Moreover, the kernel (>2.4.26) allows statically associating the elaboration of all the packets, which have to be transmitted or which have been received by a network interface, to a single CPU. This is a feasible solution to reduce concurrency effects, like cache contentions/invalidates due to locks, counters and ring buffer access, which can heavily decrease the average useful CPU cycles. By the way, as shown in [1], when the forwarding process involves two network interfaces associated to different CPUs, the packet descriptors, after that the routing decision has been taken and the egress interface selected, have to change the associated CPU, and the concurrency effects decrease the useful CPU cycles and give raise to a heavy performance reduction. Thus, in some particular cases (e.g., when it is quite likely that packets cross interfaces associated to different CPUs), increasing the number of CPUs may even decrease the overall forwarding

performance. For these reasons, to maximize the forwarding performance, it is reasonable to bind the network interfaces that exchange high amounts of packets among them to a single CPU.
Moreover, for what concerns the control plane, it is very difficult to populate a list of control functionalities that have to be probably activated in the OR. In fact, except the classical IP routing protocols, it is reasonable to suppose that the control plane functionalities to activate on a router mostly depend on the whole network scenario. For example, it is quite difficult to fix how many and what kind of functionalities and of mechanisms (e.g., Flow Access Control, Dynamic Bandwidth Allocator, and so on) an OR has to activate to effectively support the QoS.
As to the control plane, it is reasonable to suppose that the OR has to support different types of functionalities: from the implementations of classical routing protocols (RIP, OSPF, BGP, etc.), which are often used by applying well-known tools like Zebra [11], Quagga [12] and Xorp [13][14], to the ones related to monitoring modules and to QoS management applications (e.g., bandwidth allocations and flow access control). The support to all these functionalities may result in a quite high number of processes, every so often scheduled and activated, which as a whole may represent a non-negligible computational load for the OR.
Note that our main objective is not to evaluate the performance of the control plane, whose functionalities and performance analysis require both a heavy effort and the usage of sophisticated tools, to emulate large and complex network environments, but to analyze how its presence may influence the forwarding performance. For these reasons, in our tests we have decided to not take explicitly into account the real OR control plane applications, but to use a set of dummy processes to emulate them by performing a reasonable set of operations (i.e., integer and floating point calculations, memory allocations, disk writing, etc.). Thus, we have decided to use a very "CPU hungry" set of always active dummy control processes that can give us a reasonable idea of the OR performance in a worst case.

## 3 Performance tuning

As shown in Section V and reported in [1], the computational capacity power appears to be the only real bottleneck in the packet forwarding process of our OR architecture. In this context, "computational power" means both the software architecture efficiency and the characteristics of some hardware components, like the CPUs, the North Bridge chipset and the RAM banks. In fact, while for medium-large sized datagrams the OR achieves the full Gigabit/s capacity, the maximum throughput is limited to only about 400K packets per second in the presence of an ingress flow composed by 64 byte-sized datagrams, with a standard 2.5 version of the Linux kernel.
To maximize the forwarding performance, a reduction of the computational complexity of the OR software architecture is clearly needed. Thus, in this respect, we have performed the tuning of some driver and kernel parameters and introduced some interesting architectural refinements.

For what concerns the driver parameter tuning, we have to take into account that many recent network adapters [6] allow to change the ring buffer dimension and the maximum interrupt rates. Both these parameters have a great influence on the NAPI performance. In this respect, [10] shows that the interrupt rate of network adapters should not be limited in NAPI kernels and that ring buffers should be large to prevent the packet drops in the presence of bursty traffic. Note that, as a too large ring buffer may causes high packet latencies, we have chosen to set the ring buffer sizes to the minimum value that permits to achieve a good performance level.
For what concerns the kernel parameters, we have decided to over-dimension all the internal buffers (e.g., IP layer and egress buffers), to avoid useless internal drops of already processed packets. Another important parameter to tune is the ***quota*** value that fixes, in the NAPI mechanism, the number of packets that each device can elaborate at every polling cycle.
It is also possible to act on some specific 2.5 kernel parameters, by adapting them to the specific networking usage: for example, the profiling results in [1] show that the kernel scheduler operations employ about 4-5% of the overall quantity of computational resources uselessly. To avoid this CPU time waste, the OS scheduler clock frequency should be decreased: by reducing its value to 100 Hz, the forwarding rate improves of about 20K packets per second.
The rationalization of memory management is another important aspect: as highlighted in the profiling results of Section V, a considerable part of the available resources is used in the allocation and de-allocation of packet descriptors (memory management functions). Reference [16] proposes a patch that allows to recycle the descriptors of the successfully sent packets: the basic idea is to save CPU resources during the receive NAPI operations, by reusing the packet descriptors inside the completion queue. The use of this patch can again improve the performance.
Summarizing, our optimized NAPI 2.5.75 kernel image includes the descriptor recycling patch and the 5.1.13-k1 version of e1000 driver, and it has been configured with the following optimized values:

I) the Rx and Tx ring buffers have been set to 512 descriptors;
II) the Rx interrupt generation has not been limited;
III) the egress buffer size for all the adapters has been dimensioned equal to 20,000 descriptors;
IV) the NAPI ***quota*** parameter has been set to 23 descriptors;
V) the scheduler clock frequency has been fixed to 100 Hz.

## 4 Testbed and measurement tools

The OR performance can be analyzed by using both internal and external measurement methods. The external measures can be performed by using both hardware and software tools, which are outside the OR, and which usually provide global performance indexes, such as, for the forwarding process, the throughput or the maximum delay. Internal measures are obtained by using specific software tools (called profilers) placed inside the OR, which are able to trace the percentage of CPU utilization for each software modules running on the node.

Internal measurements are very useful for the identification of the architecture bottlenecks. The problem is that many of these tools require a relevant computational effort that perturbs the system performance, and that makes the results not meaningful. In this respect, their correct selection is a strategic point. We have verified with many different tests that one of the best tools is Oprofile [17], an open source code that realizes a continuous monitoring of system dynamics with a frequent and quite regular sampling of CPU hardware registers. Oprofile allows evaluating, in a very effective and deep way, the CPU utilization of each software application and each single kernel/application function running in the system with a very low computational overhead.

For what concerns the external measurements, to test the forwarding performance we have chosen an open source software-benchmarking code running on PC. More in particular, since all the well-known and commonly used traffic generation applications (like NetPerf [18] or Rude&Crude [19]) do not achieve the scalability level needed to generate and to measure high packet rates, we have used a kernel level tool. It allows a high performance level, much higher than the classical kind of software generators and measurers. The latter are simpler and they work nearer to the hardware, but obviously they have a lower flexibility level. Most of the kernel level benchmarking codes are based on the same idea: the creation in the reserved kernel space of one or more packet patterns, and their recycle for multiple transmissions. Among the different ones available, (we can cite Kernel Generator [20], UDPGen [21]) we have chosen to use the Click environment with some generation and measurement modules [10].

Thus, by using the previously cited tools, we have defined a testbed composed by the OR itself and some other PCs dedicated to traffic generation and measurement. As shown in Fig. 4, we have also used a 3Com Office Connect Gigabit Switch to aggregate the traffic generated by 3 PCs on one or more router interfaces and 1 PC to measure the output traffic.

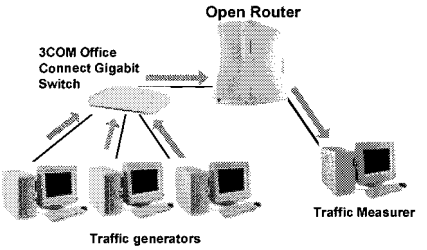



**Fig. 4.** Scheme of the testbed used for the OR performance evaluation.

## 5 Numerical Results

In this section, some of the numerical results that were obtained are shown. In particular, we have decided to report the performance evaluation results in terms of external and internal measurements for both a single processor and an SMP OR

software architecture. We have performed several tests for each selected OR configuration, with and without the activation of the control plane dummy processes, by increasing the traffic offered load. Since, as outlined in [1], the computational capacity is the main bottleneck of the OR architecture, we have chosen to perform all the tests by using a traffic flow with 64 Byte sized datagrams.

Concerning the results reported in this Section, we show two types of indexes: external throughput (i.e., the throughput crossing the OR), and profiling information, represented by the CPU percentage used by the different kernel parts. In particular, about this last type of results, all the kernel functions have been grouped in few homogeneous sets: CPU idle, scheduler, memory management, IP processing, NAPI, Tx API, IRQ routines, Ethernet processing, Oprofile and Control Plane.

Thus, in the first test session we have used a single processor running a 2.5.75 kernel with different optimization levels: a standard version, a standard version with the tuning of driver parameters, and a version that includes the parameter tuning and the descriptor recycling patch.

As shown in Fig. 5, when the control plane processes are not running, the applied optimizations allow to nearly double the maximum throughput and to achieve a forwarding rate of about to 720K packets per second (that corresponds to about half of the Gigabit capacity). The presence of active control plane processes reduces the maximum throughput obtainable with all the three adopted versions in a considerable way. In particular, Fig. 6 shows that, in such environment, the three kernel versions achieve a maximum throughput value nearly equal to 315K, 415K and 500K packets per second.

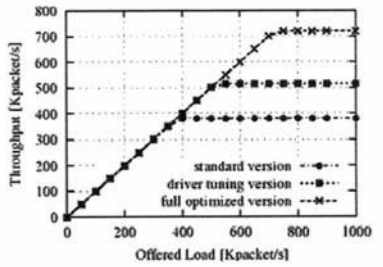



**Fig. 5**. Maximum throughput versus traffic load for different versions of single CPU kernel.

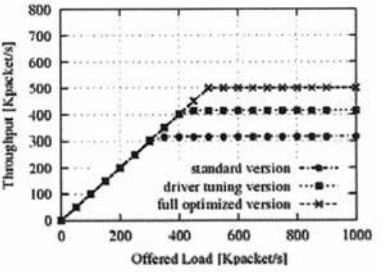



**Fig.6**. Maximum throughput versus traffic load for different versions of single CPU kernel in the presence of active control plane processes.

Figs 7, 8, and 9 report the profiling results of the tests without the control plane, while in Figs 10, 11 and 12 the ones including it are shown. In all these tests, the CPU resources used by the forwarding process (IP Processing, NAPI, Tx API, Ethernet Processing and Memory) increase their computational weight proportionally to the forwarding rate, while the IRQ management operations show a decreasing behaviour, which depends on the characteristic of NAPI kernels; when the traffic load offered to the ingress interfaces increases, the NAPI passes adaptively from an interrupt mechanism to a polling one, by reducing the interrupt rate.

Fig. 8 outlines that the driver parameter tuning causes an increase of IRQ management contribution. This effect is a clear indication that the driver parameter

tuning produces a more aggressive behaviour of the network interfaces, which try to activate the kernel network stack more frequently.
Moreover, the descriptor-recycling patch used in the fully optimized kernel version allows to reach the best forwarding performance, by reducing to zero the memory management computational weight (that reaches, in the other kernel versions, nearly 55% of the available CPU resources).

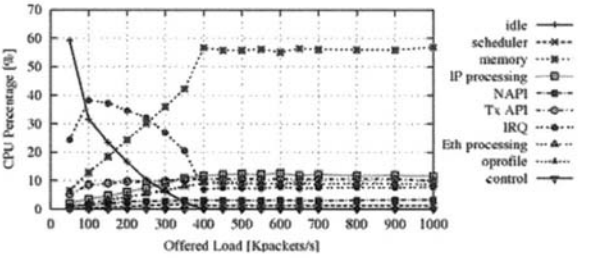



**Fig. 7**. Profiling results of the single processor standard kernel version without any processes active in the control plane. The forwarding rate is shown in Fig. 5.

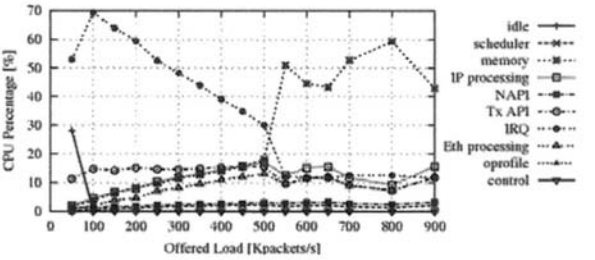



**Fig. 8**. Profiling results of the single processor kernel version with the driver parameter tuning and without any processes active in the control plane. The forwarding rate is shown in Fig. 5.

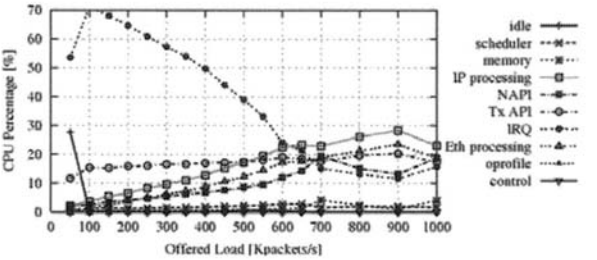



**Fig. 9**. Profiling results of the single processor fully optimized kernel version (that includes the driver parameter tuning and the descriptor-recycling patch) without any processes active in the control plane. The forwarding rate is shown in Fig. 5.

Figs 10, 11 and 12 show that, when there are active processes in the control plane, the forwarding operations appear to have a very similar behaviour, with a slightly reduced computational weight with respect to those in Figs. 7, 8 and 9. For what concerns the control plane, the active processes tend to use few computational resources in the

presence of a high IRQ rate, while, when the OR gets saturated, they become stable at a computational weight of about 30-35% of the available resources.

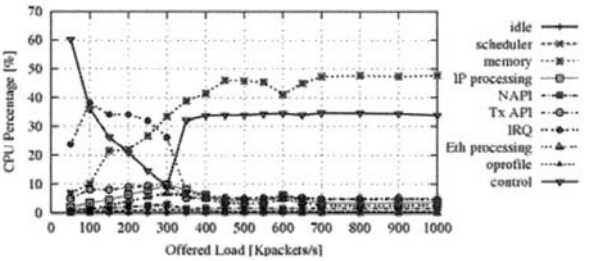



**Fig. 10**. Profiling results of the single processor standard kernel version without any processes active in the control plane. The forwarding rate is shown in Fig. 6.

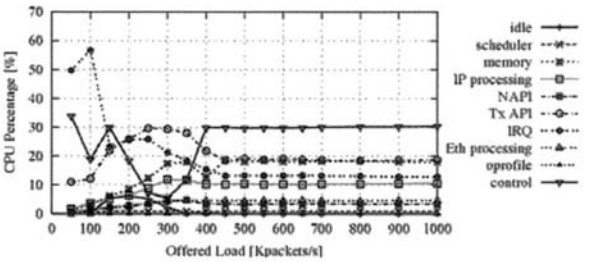



**Fig. 11**. Profiling results of the single processor kernel version with the driver parameter tuning and without any processes active in the control plane. The forwarding rate is shown in Fig. 6.

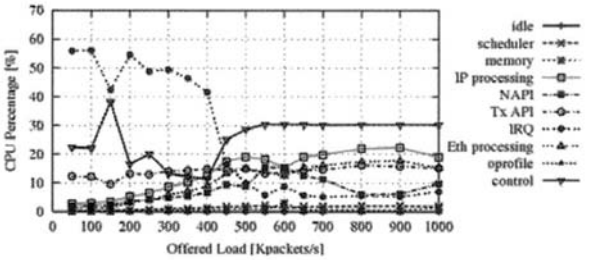



**Fig. 12**. Profiling results of the single processor fully optimized kernel version (that includes the driver parameter tuning and the descriptor-recycling patch) without any processes active in the control plane. The forwarding rate is shown in Fig. 6.

In the second test session we have used a SMP version of the optimized 2.5.75 Linux kernel (that includes the driver parameter tuning and the descriptor-recycling patch). In particular, we have chosen to use the following three different kernel configurations:

- no CPU assignments: the elaboration of both the packets received or transmitted by the different network interfaces and the control plane processes is not bound to a specific CPU. Note that, when there are no CPU-NIC bindings, the Linux kernel uses the same CPU to process all the packets.

- NICs on a single CPU: the elaboration of the received or transmitted packets is bound to CPU 0, while the control plane processes run on CPU 1.
- NICs on different CPUs: the elaboration of the received packets is bound to CPU 0, while the transmitted ones are processed by CPU 1. The control plane processes are not bound on a single CPU.

Figs. 13 and 14 report the maximum throughput values obtainable respectively without and with the active dummy processes on the control plane. Note that, when the forwarded packets have to be elaborated by two different CPUs (i.e., 3rd configuration), the memory concurrency management becomes critical and the performance of the forwarding process collapse. Moreover, with such configuration the throughput increases of about 15K packets per second when the control plane processes are active: in fact, the presence of other running processes in the system seems to reduce the concurrence of the two CPUs for the kernel reserved memory, where all the packet descriptors are located.

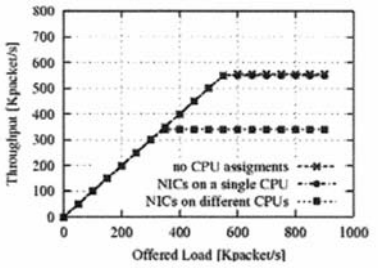



**Fig. 13**. Maximum throughput versus traffic load for different configurations of the SMP Linux kernel.

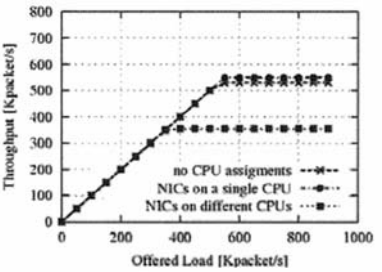



**Fig.14**. Maximum throughput versus traffic load for different configurations of the SMP Linux kernel in the presence of active control plane processes.

Figs. 15, 16 and 17 report the profiling results obtained without control plane, while Figs. 18, 19 and 20 report the profiling results of the tests while such processes are active with the three kernel configurations.

By comparing the results obtained with the first configuration (Figs. 15 and 18), we can note how the operation sets seem to have a behaviour almost similar to the single processor case: for example, also in this case, when the IRQ management functions increase their CPU occupancy, the control plane processes lower their performance. This effect is avoided by using the 2nd kernel configuration: the results in Figs. 19 do not outline any decay of the performance of control processes. Moreover, this SMP configuration seems to be the only feasible way to make the performance of the forwarding process uncorrelated with the one of the control plane.

By observing the profiling results of the 3rd configuration (Figs. 17 and 20), we can note that the CPU utilization of memory management function rises with respect to the other SMP case: as previously sketched, this particular effect is caused by a more critical memory management in the forwarding process, due to the concurrency between the CPUs to access the kernel reserved memory. Moreover, also the control plane processes seem to suffer low CPU resources.

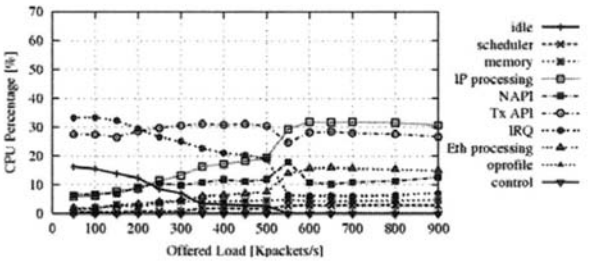



**Fig. 15**. Profiling results of the SMP fully optimized kernel version (that includes the driver parameter tuning and the descriptor recycling patch) with the 1st configuration and without any processes active in the control plane. The forwarding rate is shown in Fig. 13.

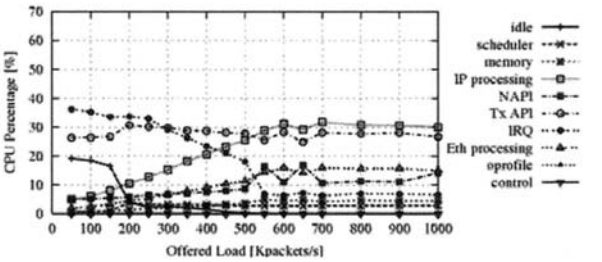



**Fig. 16**. Profiling results of the SMP fully optimized kernel version (that includes the driver parameter tuning and the descriptor recycling patch) with the 2nd configuration and without any processes active in the control plane. The forwarding rate is shown in Fig. 13.

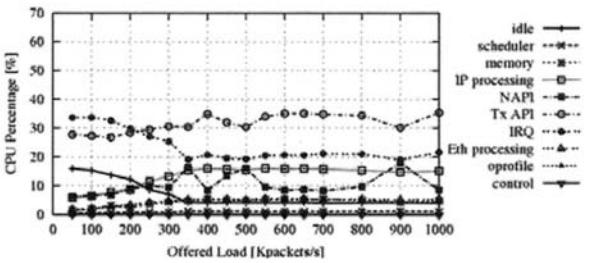



**Fig. 17**. Profiling results of the SMP fully optimized kernel version (that includes the driver parameter tuning and the descriptor recycling patch) with the 3rd configuration and without any processes active in the control plane. The forwarding rate is shown in Fig. 13.

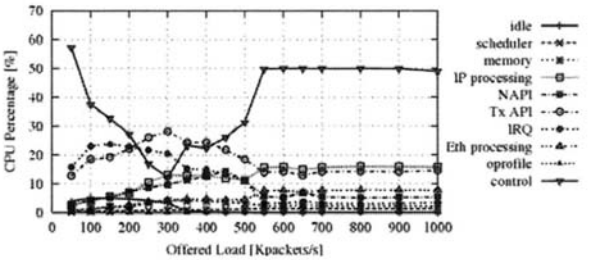



**Fig. 18**. Profiling results of the SMP fully optimized kernel version (that includes the driver parameter tuning and the descriptor recycling patch) with the 1st configuration and with the processes active in the control plane. The forwarding rate is shown in Fig. 14.

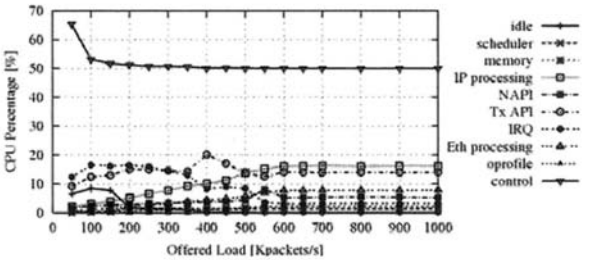



**Fig. 19**. Profiling results of the SMP fully optimized kernel version (that includes the driver parameter tuning and the descriptor recycling patch) with the 2nd configuration and with the processes active in the control plane. The forwarding rate is shown in Fig. 14.

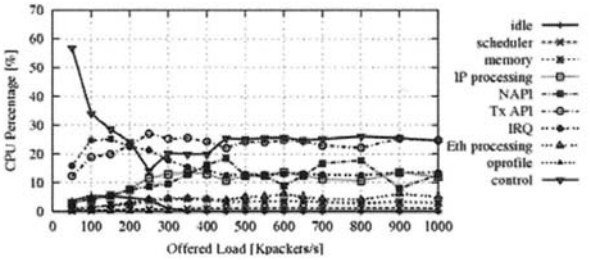



**Fig. 20**. Profiling results of the SMP fully optimized kernel version (that includes the driver parameter tuning and the descriptor recycling patch) with the 3rd configuration and with the processes active in the control plane. The forwarding rate is shown in Fig. 14.

## 6 Conclusion

The main contribution of this work has been reporting the results of a deep activity of optimization and testing realized on a PC Open Router architecture based on Linux software, and, more in particular, based on Linux kernel 2.5. The main objective has

been the performance evaluation (with respect to packet forwarding) of an optimized OR, both with external (throughput) and internal (profiling) measurements.
The obtained results show that a single processor OR can achieve interesting performance for what concerns the forwarding process, but, in the presence of active processes in the control plane this performance can notably decrease. The results obtained with the SMP forwarding kernel show that, by binding the forwarding process and the control plane functionalities to different CPUs, the OR can reach a maximum throughput nearly equal to 550K packets per second, without any interference (in terms of performance) with the control plane processes.

## Reference


1. Bolla, R., Bruschi, R.: A high-end Linux based Open Router for IP QoS networks: tuning and performance analysis with internal (profiling) and external measurement tools of the packet forwarding capabilities. Proc. of the 3rd International Workshop on Internet Performance, Simulation, Monitoring and Measurements (IPS MoMe 2005), Warsaw, Poland (2005) pp. 203-214
2. Comer, D.: Network Processors: Programmable Technology for Building Network Systems. The Internet Protocol Journal Vol. 7(4) (2004) pp. 2-12.
3. Bianco A., Finochietto, J. M., Galante, G., Mellia, M., Neri, F.: Open-Source PC-Based Software Routers: a Viable Approach to High-Performance Packet Switching. Proc. of the 3rd International Workshop on QoS in Multiservice IP Networks (QoS-IP 2005), Catania, Italy (2005) pp. 353-366
4. EURO: University Experiment of an Open Router, http://www.diit.unict.it/euro
5. Gray, P., Betz, A.: Performance Evaluation of Copper-Based Gigabit Ethernet Interfaces. Proc. of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02), Tampa, Florida (2002) pp. 679-690
6. The Intel PRO 1000 XT Server Adapter, http://www.intel.com/network/connectivity/products/pro1000xt.htm
7. The Click! Modular Router, http://www.pdos.lcs.mit.edu/ click/
8. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M. F.: The Click modular router. ACM Transactions on Computer Systems Vol. 18(3) (2000) pp. 263-297
9. Bianco, A., Birke, R., Bolognesi, D., Finochietto, J. M., Galante, G., Mellia, M.: Click vs. Linux: Two Efficient Open-Source IP Network Stacks for Software Routers. Proc. of the IEEE Workshop on High Performance Switching and Routing (HPSR 2005), Hong Kong (2005)
10. Salim, J. H., Olsson, R., Kuznetsov, A.: Beyond Softnet. Proc. of the 5th annual Linux Showcase & Conference, Oakland CA (2001)
11. Zebra, http://www.zebra.org/
12. Quagga Software Routing Suite, http://www.quagga.net
13. Xorp Open Source IP Router, http://www.xorp.org/
14. Handley, M., Hodson, O., Kohler, E.: XORP: an open platform for network research. ACM SIGCOMM Computer Communication Review Vol. 33(1) (2003) pp. 53-57
15. Cox, A.: Network Buffers and Memory Management. Linux Journal (1996), http://www2.linuxjournal.com/ lj-issues/issue30/1312.html
16. The descriptor recycling patch, ftp://robur.slu.se/pub/Linux/net-development/skb_recycling/
17. Oprofile, http://oprofile.sourceforge.net/news/
18. NetPerf, http://www.netperf.org/netperf/NetperfPage.html

19. Rude&Crude, http://rude.sourceforge.net/
20. UDPGen, http://www.fokus.gmd.de/research/cc/berlios/employees/sebastian.zander/private/udpgen/
21. The Linux Documentation Project, http://www.tldp.org/tldp-redirect.php?url=/HOWTO/Kernel-HOWTO.html

http://www.springer.com/978-0-387-29811-5

Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements

Davoli, F.; Palazzo, S.; Zappatore, S. (Eds.)

2006, XII, 232 p. 258 illus., Hardcover
ISBN: 978-0-387-29811-5