A Divert Mechanism for Service Overlays

Akihiro Nakao, Larry Peterson and Mike Wawrzoniak
{nakao,llp,mhw}@cs.princeton.edu

Department of Computer Science
Princeton University

Abstract

Shared overlay nodes such as the ones in PlanetLab [5] enabled a new perspective on overlays, in which a user can subscribe to arbitrary overlays, each of which implements some network service on behalf of the user's applications. We have designed and implemented a general-purpose mechanism to transparently divert packets belonging to designated applications onto service overlays, and we demonstrated how our mechanism can be used to divert packets from applications like `http, ssh, sftp` onto a RON-based [1] routing overlay.

1 Introduction

Overlays are gaining popularity as a vehicle for deploying network services. Examples include resilient transport [1], distributed object location [6, 7, 10], peer-to-peer storage [4, 8], and multicast [2, 3]. Most of these services behave in the same way: participants collectively and cooperatively implement some service for the benefit of all overlay members.

This paper proposes a new perspective on such overlay networks. Provided there are multiple *service overlays* that implement their own network services on shared overlay nodes, we let users subscribe to arbitrary service overlays in a way that is transparent to applications they are running.

In our design, we made a clear distinction between *desktop* and *overlay* nodes. A desktop is a user's machine that subscribes to a service overlay. Desktops are usually under full control of the user. All privileged operations, such as loading kernel modules and setting up firewall rules, are allowed on desktop machines. On the other hand, overlay nodes are shared, protected, and restricted, intermediate resources, where privileged operations are prohibited, or modified to be restricted or protected in some way. We made use of protected RAW socket machanism avaliable on our overlay nodes.

This paper explores the idea of transparently diverting designated packets onto arbitrary service overlays. By implementing an example routing overlay service based on RON [1] on PlanetLab [5], we have identified a general-purpose mechanism that allows users to connect their desktops to arbitrary service overlays, with both client and server applications unaware of the existence of the service overlay. In addition, by turning RON into an overlay service, we are able to scale RON to a larger number of client nodes, and allow thin clients to access the service as well.

2 Example Service Overlays

Routing in the today's Internet has several limitations: it is slow to switch to a redundant path in the event of failure, and it fails to take application requirements (e.g., bandwidth, latency, loss) into account when selecting routes. RON attempts to solve these problems by forming a small clique of end-system nodes, and measuring link properties among them frequently.

We propose a small variant of this model, in which the RON functionality is moved onto a set of dedicated overlay nodes, such as PlanetLab [5] and each end-system associates itself with a nearby overlay node. This effectively allows RON to aggregate traffic on behalf of a larger collection of end-systems. It also means RON can support thinner end-systems, for example mobile nodes that cannot afford the measurement burden RON imposes.

Although we used RON as a routing algorithm in our routing service overlay, our diverting mechanism is generic enough to accommodate other routing service overlays that use different routing algorithms, such as end-system multicast or DHT-based routing. Moreover, the diverting mechanism could be applicable to many other types of service overlays that take advantage of intermediate overlay nodes to perform transparent add-on services (e.g., video

transcoding) along an end-to-end path.

3 Divert Mechanism

This section describes our general scheme to divert packets through service overlays by illustrating the case of a routing overlay. Although the discussion may sometimes sound specific to routing service overlays, we believe the diverting scheme is applicable to general service overlays. Figure 1 gives a high-level view of our diverting mechanism.

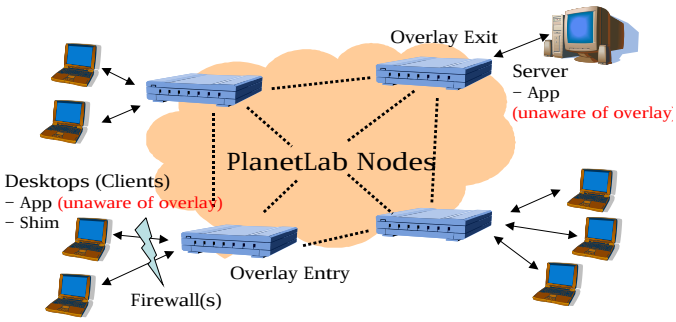

Figure 1: Overview

A shim running on the desktop intercepts packets that the client application sends, and diverts them to an overlay entry node. This node then routes the packets through the overlay to an exit node, which forwards the packet to the application server. We also intercept the returning packets from the server, forward them through the overlay, back to the desktop shim, and finally to the client application. Note that client and server applications are both unaware of the routing service overlay. Although desktops may be behind firewalls, we assume that overlay nodes all lie in the Internet.

3.1 Architecture

As shown in Figure 2, our divert architecture has two components: a *desktop shim (DS)* and a *service access module (SAM)*. The DS runs as a separate process on the desktop where the client application is running, and upon request, is dynamically bound to the client application. A SAM module is linked into the service overlay code running on each overlay node. For a given session, the SAM at the entry and exit overlay nodes play a role. We define a *service access protocol (SAP)* that is used by the DS to communicate with the SAM on the entry node. Currently, SAP defines two channels: a *lookup* control channel and data *forwarding* channel.

The DS performs two tasks: (1) it intercepts packets

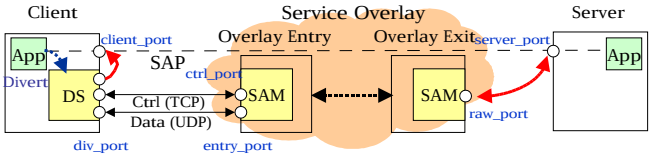

Figure 2: Divert Architecture

sent by the client program and diverts them onto the overlay, and (2) it receives packets from the overlay and makes them look like they arrived directly from the server. Note that only the DS is aware of the overlay, that is, the divert mechanism is totally transparent to the application.

As shown in Figure 3, we have implemented DS on Linux 2.4.19 kernel, using kernel modules `netfilter`, `ip_queue` and user programs `divmod`, `ipq_lib`. Central to DS software is `divmod`. The DS can be implemented on other platforms as well, for example in BSD using `divert socket`.

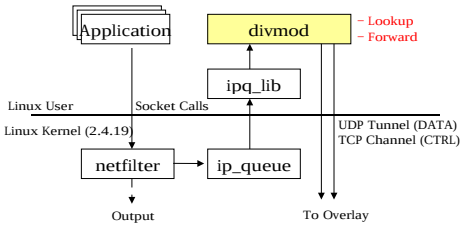

Figure 3: Desktop Shim on Linux 2.4.19

The DS is configured to bind an application, as identified by a (`protocol`, `server_port`) pair to a particular SAM, as identified by an (`entry_ip`, `ctrl_port`) pair. In our prototype, we use `iptables` facility to bind the DS to the client program. When the DS sees the first packet belonging to a new session, it blocks the subsequent packets of that session and sends a lookup control message to the corresponding SAM. This lookup control channel is implemented using TCP, and the control message includes the first 128 bytes of the intercepted message.

The SAM at the overlay entry determines whether it can process this session based on this information, and if it can, replies with a session-specific (`entry_ip`, `entry_port`) pair. In our prototype, the SAM at the overlay entry makes this decision if the destination address is on the same network as a known overlay node, denoted as (`exit_ip`). If the SAM at the overlay entry cannot help in this case, it informs the DS of this fact and all subsequent packets are delivered directly over the Internet. If the lookup operation was successful, it forwards the first packet, and then

any subsequent packets over a UDP tunnel using forward data channel to (`entry_ip, entry_port`). We use UDP data channel, not TCP, since the end-to-end communication may be using TCP and we would avoid the complexity of having another TCP connection nested inside a TCP connection.

3.2 Step-by-Step Packet Trace

Forwarding packets through the overlay requires doing network address translation (NAT) at overlay entry and exit nodes. Figure 4 summarizes how an example TCP packet is transferred between client application and server application. Each component sends and receives data packets both in the *forward* direction (from client to server) and in the *reverse* direction (from server to client). We now walk through the process step by step.

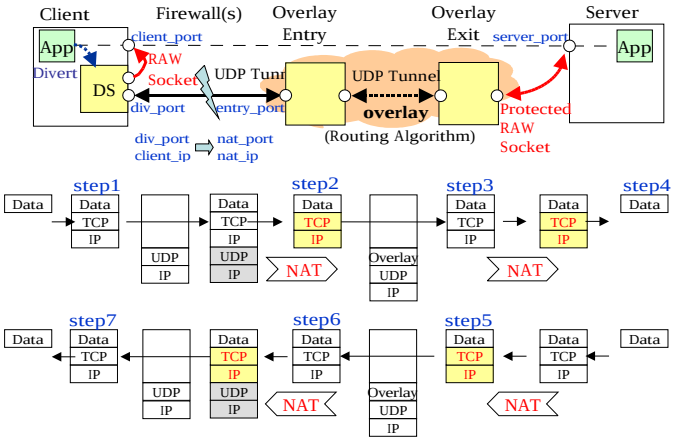

Figure 4: TCP packets forwarding

Step 1: Client (forward)

A client application sends packets whose header includes client address (`client_ip, client_port`) as the source address, server address (`server_ip, server_port`) as the destination address and protocol (`protocol`). The DS selectively intercepts designated packets and redirect them to the overlay at UDP port `entry_port`. The source port for these tunneled messages is `div_port`.

Step 2: Overlay Entry (forward)

At the overlay entry node, the SAM receives packets at the UDP port, `entry_port` from the address `nat_ip, nat_port`, which are translated from the DS's address `client_ip, div_port` as a result of NAT(s) at any firewall(s) that happen to be between the desktop and the overlay entry node. Thus, `nat_ip, nat_port` may or may not be the same as `client_ip, div_port`, depending on the existence of firewall(s). As we describe later, we need to keep state for each session at the overlay entry and the overlay exit, so we need to define a unique session ID (`session_id`). We use the following 5-tuple as session ID:

```
session_id = <nat_ip, nat_port, server_ip
              server_port, protocol>
```

Since we need to use this session ID at the overlay exit too, we need to carry `nat_ip, nat_port` along with the packet, perhaps in the overlay header. In order to avoid these extra bytes, we decided to do the first NAT and rewrite the original packet header as follows. Note this NAT does not require root privilege since everything is encapsulated in UDP or the overlay header.

```
src_ip   = nat_ip
src_port = nat_port
```

The overlay entry records the following information in a hash map (`entry_map`):

```
entry_map:
 session_id => <client_ip, client_port,
                exit_ip, timestamp>
```

Since we use a UDP tunnel between the desktop and the overlay entry, we can find the original client address (`client_ip, client_port`) deep in the original packet. `exit_ip` is selected for the first packet of a new session, and stored in the hash map for subsequent packets. `timestamp` is the time when the last packet of this session was forwarded.

After we strip the divert header from the packet, we attach the overlay header, and send the packet to overlay internal nodes according to some routing algorithm like RON. Note that `entry_ip` is sent to the overlay exit in the overlay header.

Step 3: Overlay Exit (forward)

At the overlay exit, the local SAM receives packets from the overlay internal nodes. After stripping off the overlay header, it forwards these packets to the server application. A complication here is that we need to intercept packets returning from the server application transparently and forward them through our overlay again. Decomposing the end-to-end connection would break the end-to-end semantics between applications, which is against our design policy that our divert mechanism be totally transparent to the applications. Therefore, we decided to utilize *protected RAW socket (PRS)* available on overlay nodes, and to perform another NAT as follows.

```
src_ip   = exit_ip
src_port = raw_port
```

When we see the first packet of a new session, we open a PRS (raw_sock) and use it for the subsequent packets. PRS differs from the original RAW socket as follows. First, it does not require root privilege to use PRS. Note that the NAT at Step 3 actually needs to edit the header of a bare IP packet. Therefore we would have to need root privilege to send out the packet, if we were using the original RAW socket, not PRS. Second, a PRS user not only specifies a protocol number but also a port number. Once the user opens a PRS with a specific protocol and port number as a demux key, nobody else can intercept the packets with that demux key. Third, we cannot spoof the source address of the packet departing an overlay node via PRS. Finally, PRS grabs the original (not a copy of) packet.

The overlay exit needs to record the following information in two index hash maps. The first hash map (exit_map1) associates a session ID (session_id) with a PRS number (raw_sock) and its port number (raw_port). The second hash map (exit_map2) associates a PRS number with a session ID, IP address of entry, and timestamp. exit_map1 will be used to check if we can reuse raw_sock and raw_port for session_id. exit_map2 will be used to classify and edit the returning packets from the server in Step 5.

```
exit_map1:
session_id => <raw_sock, raw_port, timestamp>
exit_map2:
raw_sock => <session_id, entry_ip, timestamp>
```

Step 4: Server

The server application observes the other end of communication is the overlay exit. In response to the received packets, the server application sends back the reply packets whose header includes the following information.

```
src_ip   = server_ip
src_port = server_port
dst_ip   = exit_ip
dst_port = raw_port
```

Step 5: Overlay Exit (reverse)

There are two things that happen in reverse delivery at SAM on the overlay exit node: packet classification and packet editing. As we discussed in Step 3, since PRS demultiplexes packets with specific protocol and source port numbers, we receive the packets from the server at the corresponding raw_sock. Since we use the same PRS raw_sock both for sending packets in the Step 3, and for receiving packets here, we can retrieve the information about the session by consulting exit_map2.

Once the SAM classifies an arriving packet, it edits their headers (NAT). Since everything will be encapsulated in UDP from now on, until we really transmit the packet to the client application in Step 7, we do not need to use RAW socket yet, and NAT does not require root privilege either. After retrieving session_id, entry_ip from exit_map2, we just rewrite the IP header as follows.

```
dst_ip   = nat_ip
dst_port = nat_port
```

This packet is then encapsulated in the overlay header and sent back to the overlay entry via a UDP tunnel.

Step 6: Overlay Entry (reverse)

At the overlay entry node, the local SAM gets packets from overlay internal nodes. After stripping off the overlay header, we retrieve the session information by consulting entry_map, and edit the packet header as follows (NAT). This NAT is necessary to make the packet look like destined to the address of client application at Step 7.

```
dst_ip   = client_ip
dst_port = client_port
```

It then attaches a divert header to the packet and relays it back to the DS's NAT-ed address (nat_ip, nat_port) via UDP tunnel. This NAT-ed address will be translated by the firewall(s) to the DS's private address (client_ip, div_port) so the packets may reach the DS program.

Step 7: Client (reverse)

Finally, the DS receives packets from the SAM at the overlay entry. It strips off the divert header and sends the packet to the client application via the original RAW socket. Note that the packet already has the correct address (client_ip, client_port) as a result of the NAT at Step 6.

3.3 Remarks

As explained above, the overlay entry and the exit nodes keep *soft-state* about sessions. Figure 5 summarizes the mapping between session ID and overlay parameters. In our design, we want to avoid out-of-band signaling as much as possible, since overlay nodes may come and go

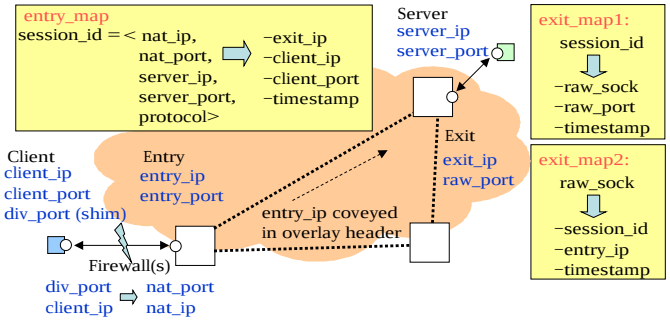

Figure 5: Per-Flow Soft-State Mapping

dynamically. We add a session entry when we see the first packet of a new session, and evict a session entry after the session has been idle for relatively long time. We also consider protocol-based optimization. For example, in case of TCP protocol sessions, we invalidate a session entry shortly after we detect TCP FIN or TCP RST. On the other hand, in the case of UDP sessions, it would probably be fine to invalidate the session entry quickly. Note that our mapping scheme is resistant to the loss or the replacement of entry node, but not that of exit node. This asymmetry results from the complication that we need to receive packets returning from the server onto the overlay transparently.

4 Conclusion

Our contributions in this paper are two-fold. First, we have designed and implemented a general divert mechanism. Second, we have demonstrated a RON-based routing service overlay using RON on PlanetLab. We are able to run several applications, such as `http, ssh, sftp`, on top of the resulting system.

We note that our divert mechanism resembles Detour [9] in several ways, although no implementation details are given for Detour. Detour uses in-kernel routing architecture to support alternate route by IP-IP tunnels. Detour directs its outbound traffic to the nearest Detour router, these packets are forwarded along tunnels, and then exit at a point close to the destination. Interestingly, Detour points out the complication of NAT to get responses on Detour routers again.

Although our early stage implementation has already proved the validity of the scheme, we do see the following challenges. First, since we use PRS to splice the end-to-end connection, port numbers became limited shared resources. We also use soft-state mappings on overlay entry and exit per session, which may consume a fair amount of memory. How to relinquish these per-session resources is one of the challenges. Second, we have very preliminary scheme to look up overlay entry and exit for a given session based on autonomous system numbers. We are likely to need a more sophisticated algorithm for this. Although designing new routing overlays was not our main focus, we realize that addressing scalability problem of routing overlays beyond traffic aggregation is an interesting research problem.

We recognize that emerging overlay services may introduce the need to extend the service access module or the desktop shim with more elaborate translation steps beyond packet manipulation and redirection. We believe our model can be extended to support these services by introducing translation plug-ins in SAM and DS, which will be addressed in our future work.

References

[1] D. Andersen, H. Balakrishnam, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, October 2001.

[2] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of the ACM SIGCOMM Conference*, pages 1–12, August 2001.

[3] Y.-H. Chu, S. G. Rao, and H. Zhang. A Case For End System Multicast. In *Proceedings of the ACM SIGCOMM Conference*, pages 1–12, June 2000.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[5] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the HotNets-I*, 2002.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proceedings of the ACM SIGCOMM Technical Conference*, 2001.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.

[8] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-Scale Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Princeiples (SOSP)*, pages 188–201, October 2001.

[9] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: A Case for Informed Internet Routing and Transport. *IEEE Micro*, 19(1):50–59, January 1999.

[10] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, pages 149–160, 2001.