Autonomic Fail-over for a Software-Defined Container Computer Network

Chien-Yung Lee[+], Yu-Wei Lee[+], Cheng-Chun Tu[*+], Pai-Wei Wang[+], Yu-Cheng Wang[+], Chih-Yu Lin[+], and Tzi-cker Chiueh[*+]

[*]Computer Science Department, Stony Brook University
[+]Cloud Computing Center for Mobile Applications, Industrial Technology Research Institute

Abstract

The ITRI container computer is a modular computer designed to be a building block for constructing cloud-scale data centers. Rather than using a traditional enterprise data center network architecture, which is typically based on a combination of Layer 2 switches and Layer 3 routers, the ITRI container computer's internal interconnection fabric, called *Peregrine*, is a software-defined network specially architected to meet the scalability, fast fail-over and multi-tenancy requirements of these data centers. *Peregrine* uses as the underlying physical interconnect a mesh of commodity off-the-shelf Ethernet switches, and adopts a centralized network control architecture that operates these Ethernet switches as a coordinated distributed data plane. Compared with vanilla enterprise networks, *Peregrine* features a fast fail-over capability not only for network switch/link failures, but also for failures of its own control servers. This paper describes the design and implementation of *Peregrine*'s fault tolerance mechanisms, and shows their effectiveness using empirical performance measurements taken from a fully working *Peregrine* prototype under various failure scenarios.

1 Introduction

The ITRI container computer is designed to be a modular building block for constructing a *cloud data center computer*, which, in the most general form, is composed of multiple container computers that are connected by a data center network, is interfaced with the public Internet through one or multiple IP routers, and is designed as an integrated system whose hardware components such as servers and switches are stripped off unnecessary functionalities, whose resources are centrally configured, monitored and managed, and which encourages system-wide optimizations to make the best end-to-end tradeoffs. One key design decision of the ITRI container computer is using only commodity hardware, including compute servers, network switches, and storage servers, and leaving high availability and performance optimization to the system's software. Another key decision is to design a new data center network architecture from the grounds up to meet the unique requirements imposed by a cloud data center computer. We named this data center network architecture *Peregrine* [5]. This paper focuses on the design, implementation and evaluation of the fault tolerance mechanisms in *Peregrine*.

Although *Peregrine* uses commodity off-the-shelf Ethernet switches as basic building blocks, it follows a software-defined network (SDN) [4, 8] design philosophy by doing away with most of the control plane functionalities in these switches and using a centralized network control server to operate these switches, and eventually turning them into a coordinated distributed data plane. *Peregrine* chooses this centralized control plane architecture because it offers two important advantages. First, it enables *Peregrine* to make more efficient use of all physical links in the underlying network. Second, it significantly reduces the fail-over latency associated with any single network switch/link failure.

Despite various optimizations, standard Ethernet-based networks take at least a few seconds to recover from a network switch/link failure, especially for large networks, because their normal operation assumes a spanning tree overlaid on top of the physical network, and re-building this spanning tree after a failure takes time. A fail-over latency of several seconds is not acceptable in large-scale data centers that are built out of commodity hardware components, because in these data centers HW failures are not uncommon and they need to be effectively masked so as to be completely hidden from applications and their users.

The fail-over latency goal of the ITRI container computer is set to 100 ms, which is set so as to mask each network failure event as a transient congestion. To achieve this goal, *Peregrine* does away with the concept of span-

ning tree completely, and therefore does not need to rebuild anything after a failure; moreover, it pre-computes and pre-installs a contingent plan for all nodes that are affected by every possible network switch/link failure to route around the failure, thus greatly reduces the fail-over latency to the minimum. This fast fail-over strategy is made possible by the centralized control plane architecture, because it is equipped with a global knowledge of the physical network topology, its up-to-date health status, and the network flows provisioned on them.

However, a centralized control plane is in theory more brittle because it is a single point of failure, that is, any control plane failure could potentially bring down the entire network. To overcome this issue, *Peregrine*'s control servers are designed to be fully redundant and thus highly available. The interaction of the high availability (HA) mechanism of *Peregrine*'s control servers with *Peregrine*'s fail-over mechanism is complex and subtle, and requires careful considerations to every low-level detail.

Finally, because the servers used in the ITRI container computer are also of commodity grade, the failure rate of these servers is not negligibly low. To enable seamless fail-over of application VMs running on these servers, *Peregrine* informs clients that interact with failed VMs and redirects them to their backup VMs that take over.

2 Related Work

There has been extensive studies on the resiliency of conventional Internet [10,13]. These works compute a number of node or link disjoint paths between pairs of end points and switch over to its corresponding backup path upon link or switch failures. Provider Backbone Bridge Traffic Engineering (PBB-TE) swaps the B-VID value to redirect the traffic onto the pre-configured path within 50 ms under a path failure [3] . MPLS-TE [11,17] offers fast reroute functionality by redirecting encapsulated traffic to a backup path when the primary one fails. Mechanisms for monitoring and discriminating against intermittent link failures to achieve network stability are also addressed in [1,16].

Numerous data center network architectures propose the fault tolerant data plane by introducing a centralized controller. PortLand [15] employs a centralized fabric manager and relies on switches to detect and inform its centralized fabric manager when a link or switch fails. The fabric manager maintains a fault matrix with per-link connectivity and informs affected switches to re-route packets. VL2 [9] depends on OSPF to re-converge quickly and allows applications to fully use a link several seconds after it is restored, due to the conservative defaults for OSPF timers. VL2's directory server also incorporates the asynchronous replicated state machine to offer a strongly consistence based on the Paxos consensus algorithm.

SDN proposes separation and centralization of the control plane from the data plane. Most of the existing OpenFlow-based SDN proposals address resiliency at either the controller [12,20] or the data plane. Onix [12], a distributed control platform, provides coordination facilities for detecting and reacting to Onix instance failures. FlowVisor [18] partitions the underlying network and allows multiple controllers to manage their own slice of network. The data plane reliability relies on either the controller *proactively* pre-computes and pre-installs the backup paths on an OpenFlow switch [14] or *reactively* takes action upon receiving failure notifications [19]. For example, NOX [20] depends on existing switch mechanisms to determine link failures, notify NOX, and flushes the flow entries at that switch which use the failed link. However, the proactive mechanism requires installing additional flow entries into the OpenFlow switches, which has very limited TCAM entries, whereas the reactive mechanism incurs high latency. Moreover, one of the major concerns about SDN's split architecture design is the resiliency between the centralized controller and switches [19]. That is, any failure that disconnects the data plane form the control plane may bring down the entire network [2, 21]. Existing SDN proposals depends on an out-of-band control network to guarantee reachability between switches and the controller. However, the fail-over latency between controller and switches is usually at the timescale of seconds, due to the fact that the control network is running conventional distributed protocols such as spanning tree protocol (STP), IS-IS, or OSPF.

We argue that a well-architected SDN should have its fast fail-over mechanism among the data plane, the controller, and the control plane. *Peregrine* takes the first step in addressing all these three aspects using standard Ethernet switches and in-band control design.

3 Fault Tolerance Support in Peregrine

3.1 ITRI Container Computer

The ITRI container computer is physically housed in an ISO-standard 20-foot (6.096 meter) shipping container, and consists of 12 server racks lined up on both sides of the container with an access aisle in the middle, where each server rack holds up to 96 current-generation X86 CPUs and 3TB of DRAM. Twelve JBOD (Just a Bunch Of Disks) storage servers, each packed with 40 disks, are installed in the container computer. Together with the local disks directly attached to compute server nodes, the container boasts of more than 1 petabyte worth of usable disk space.

The ITRI container computer's network is a modified Clos network. Every rack contains 48 server nodes, each having 4 1GE NICs, and includes 4 top-of-rack (*TOR*) switches, each having 48 1GE ports and 4 10GE ports. There is a virtual switch inside every server node that is connected to the server node's four NICs, which in turn are connected to the four TOR switches in the same rack. The four 10GE unlinks on each TOR switch are connected to four different *regional* switches, each of which has 48 10GE ports. To improve the performance of storage accesses, each storage server has four 10GE NICs and is directly connected to four different regional switches. In total, five regional switches per rack are used in the ITRI container computer.

Peregrine [5] is the internal network for the ITRI container computer, and is built on commercially off-the-shelf Ethernet switches with most of their built-in control plane functionalities such as spanning tree protocol, source learning, flooding if unknown destinations, etc., turned off. Instead, *Peregrine* uses a centralized control plan that manages the forwarding tables of the underlying Ethernet switches. The software architecture of *Peregrine* is shown in Figure 1, and consists of a *kernel agent* that performs ARP query packet interception and transformation and is installed in the Dom0 VM of every Xen-based physical server, a *centralized directory server* (DS) that performs generalized IP to MAC address look-up, and a *centralized route algorithm server* (RAS) that constantly collects the network's traffic matrix, runs a load-based routing algorithm based on the traffic matrix, and populates the switches' forwarding tables with the computed routes. After the RAS computes routes for physical server pairs, it builds up an *inverse* map that associates every network link with all the computed routes that go through the link.

All packets from a DomU VM pass through the *Peregrine* agent in Dom0 of the corresponding physical machine. For each packet going by, the *Peregrine* agent consults with its local ARP cache with the packet's destination IP address, submits a lookup request to the DS if the cache lookup is a miss, and rewrites the packet's destination MAC address field based on the ARP look-up result from the local cache or the DS.

3.2 Centralized IP Address Resolution

Because *Peregrine* is designed to scale to a large number of physical servers using only L2 connectivity, it discourages broadcast-based protocols such as ARP (Address Resolution Protocol) and DHCP (Dynamic Host Configuration Protocol). Instead, it replaces them with a client-server architecture, where queries are directed to a dedicated server, which answers these queries by looking up its internal data structures. This design change

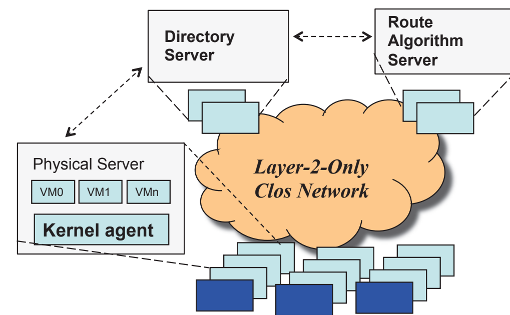

Figure 1: *The software architecture of the current* Peregrine *prototype, which consists of a kernel agent installed in the Dom0 VM of every physical machine, a centralized directory server (DS) for IP to MAC address look-up, and a centralized route algorithm server (RAS) for route computation and forwarding table population.*

is similar in spirit to how cache coherence protocols in shared-memory multiprocessor systems progressed from broadcast-based to directory-based as their scale increases.

When a user VM sends out a broadcast-based ARP query, a *Peregrine* agent running at the same physical server intercepts it, converts the query into a unicast packet and sends it to a central *directory server*, which maintains an *address resolution map* between VMs' IP addresses and their MAC addresses, and answers these transformed ARP queries. After receiving answers from the directory server, the *Peregrine* agent converts it into a legitimate ARP response packet, sends it to the original querying user VM, and caches the answers for future reuse. Therefore, not every ARP query needs to be sent to the directory server; in fact, most ARP queries are expected to be answered by the caches maintained by *Peregrine* agents.

To ensure the consistency of ARP caches, *Peregrine*'s directory server adopts a *lease-based* stateful cache coherence protocol. That is, every cached ARP query response is given a fixed lifetime, say 2 minutes, and the directory server keeps a record of which physical server caches which ARP query responses, each of which consists of an IP address and its corresponding MAC addresses (it would become clear later why multiple MAC addresses are associated with an IP address). When an ARP query response resides in a physical server longer than the fixed lifetime, it becomes invalid and cannot be used to answer ARP queries. The key design challenge in this stateful cache consistency maintenance mechanism is how to reduce the amount of state required. Suppose the maximum number of VMs in a cloud data center is 100,000 VMs, and every VM could communicate with at most N VMs, then the address resolution map in the directory server has 100,000 entries, each of which in turn contains up to N VM IDs and N timestamps of when the entry is cached in each of the N VMs. Whenever the directory server modifies an entry in its address resolution

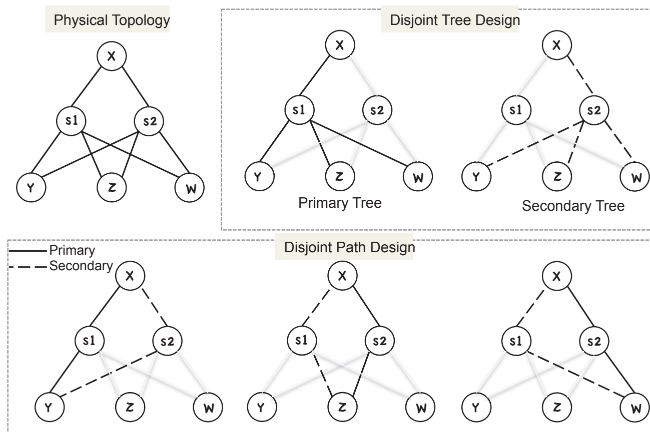

Figure 2: *An example network to illustrate the difference between a disjoint tree design (above) and a disjoint path design (below). In the disjoint path design, the primary path of one server pair (X-Z) could overlap with the secondary path of another server pair (X-Y) as long as it does not overlap with the first server pair's corresponding secondary path.*

map, it goes through the physical servers recorded in the entry, checks their timestamps to see if they expire, and sends a unicast-based invalidation notification to each of those that still hold a valid cached copy. If the number of physical servers caching an address resolution map entry exceeds *N*, the entry is flagged, and the directory server sends a broadcast-based invalidation notification instead when the entry is modified.

The additional level of indirection introduced by the address resolution map and the directory server's ability to invalidate cached ARP query responses plays a critical role in *Peregrine*, and serves multiple purposes, including scaling up the network size, redirection in VM migration, and fail-over in network switch/link failure, which we will describe in the next subsection.

3.3 Proactive Primary/Secondary Routing

When a network switch/link fails, the design goal of *Peregrine* is to route all affected network flows around the failure so that the end-to-end disruption to the communicating parties of these network flows is no more than 100 ms. To achieve this aggressive goal, for a given physical server X, *Peregrine* proactively pre-computes a *primary* and a *secondary* route from every other physical server to X, where the primary route and secondary route are node-disjoint and link-disjoint excluding the two end points, assuming the underlying physical network connectivity offers enough redundancy for such disjoint paths. Whenever a network link or switch fails, the primary routes provisioned on the failed device or link are identified, and the physical servers that are using these primary routes are notified to switch to their corresponding secondary routes. In this design, the fail-over delay of a network device/link failure thus consists of (a) the time to detect the device/link failure, (b) the time to identify affected primary routes and their source physical servers, and (c) the time to inform these affected source servers to switch from primary to secondary routes.

The first design issue is how to switch from primary to secondary routes when failures occur. Because *Peregrine* uses conventional Ethernet switches and Ethernet switches forward packets based on their destination address, the only way to send packets to a given physical server X using multiple routes is to assign multiple MAC addresses to X, each representing a distinct route to reach X. At start-up time, *Peregrine* installs pre-computed primary/secondary routes to every physical server in the switches' forwarding tables. At run time, switching from the primary to the secondary route of a given server is a matter of using the server's secondary MAC address rather than primary MAC address. Modern operating systems, including both Linux and Windows, allow multiple NICs to be bound to the same IP address, through DHCP or through user-entered commands.

Given a physical server X, all other servers that *send* packets to it form a spanning tree. When computing primary and secondary routes for X, there are two possible designs: *disjoint tree* and *disjoint path*. In the *disjoint tree* design, the system computes a primary spanning tree and a secondary spanning tree that are node-disjoint and link-disjoint from each other. In the *disjoint path* design, the system computes a primary route and a secondary route between X and *every other* server, and they are node-disjoint and link-disjoint. In the first design, all other servers that send packets to X either use the first or second spanning trees, but not both simultaneously. However, in the second design, some servers that send packets to X may use the first spanning tree, while the others may use the second spanning tree at the same time. Figure 2 shows an example that illustrates the difference between these two designs. In the disjoint tree design, the primary spanning tree rooted at node X is disjoint from X's secondary spanning tree. However, in the disjoint path design, the primary path for X-Z could overlap with the secondary path for X-Y , and the secondary path for X-W could overlap with the primary path for X-Y. Obviously the disjoint path design is more flexible than the disjoint tree design, but also requires more state to be maintained on the directory server.

The trade-off between these two designs is the amount of directory server state required and the routing flexibility. In general, the larger-granularity the unit of disjointness, the more difficult it is to successfully overlay two such units on a given physical network. Because the granularity of disjointness in the *disjoint tree* design is larger than that in the *disjoint path* design, it is more difficult to successfully compute routes for the *disjoint tree*

design than for the *disjoint path* design on the same physical network. That is, for a given physical server X, it is less likely to find two disjoining spanning trees rooted at X, than to find two disjoint paths between X and any other server. In addition to routing flexibility, the *disjoint tree* design also incurs higher collateral damage when a network switch/link failure. In other words, any failure that affects (even a slight portion of) a given server's primary spanning tree renders the entire spanning tree unusable.

Given a lookup request for a physical server X, the directory server's response to it is independent of the source issuing the lookup request in the *disjoint tree* design, but is dependent on the source in the *disjoint path* design. The additional flexibility enables the *disjoint path* design to use both spanning trees associated with a physical server simultaneously, but also requires more state to be maintained on the directory server in the second design. For the *disjoint tree* design, the directory server only needs to maintain two bits for each physical server to indicate the health status of its two disjoint spanning trees. For the *disjoint path* design, the directory server needs to maintain two bits for every other server that sends packets to every given server to indicate the health status of the two disjoint paths between them. Therefore, the amount of availability-related state on the directory server is O(M) for the *disjoint tree* design and O(M^2) for the *disjoint path* design, where M is the number of physical servers on the network. As shown in Figure 2, in the disjoint tree design, the directory server maintains two bits for X's primary tree and backup tree and a failure of any link in the primary tree triggers all other servers to switch to the backup tree, whereas in the disjoint path design, the directory server maintains two-bit health status for paths between Y to X, Z to X, and W to X. If a link between X and s1 fails, only the primary path between X and Y and the secondary paths between X and Z and between X and W are affected.

The current *Peregrine* prototype adopts the *disjoint path* design to successfully fail over as many communicating node pairs affected by a given network switch/link failure as possible. To reduce the amount of availability-related state on the directory server, *Peregrine* uses a list structure that can dynamically grow and shrink its size to record the set of physical servers with which a given physical server is currently communicating. From the analysis of several data center traffic traces [9], we assume the majority of physical servers communicate with at most N other servers, and the total amount of availability state that needs to be maintained is proportional to MN rather than M^2.

The availability bits associated with a physical server are stored in the server's address resolution map entry in the directory server, together with its timestamps associated with stateful caching. In summary, a physical server's address resolution map entry consists of the following:

- An IP address,
- Two MAC addresses, and
- A communication list of entries, each of which contains a caching timestamp, two availability bits and a primary/secondary flag for each physical server that it currently communicates with.

Every physical server, say S1, is assigned an address resolution map entry, and every other physical server that communicated with S1, say S2, is assigned an entry in S1's communication list, which indicates which of S1's two MAC addresses is the primary MAC address and whether the two paths between S1 and S2 are available or not. When another server, say S3, just starts to communicate with S1, *Peregrine* inserts an entry <011> [1] to S1's communication list, meaning that the currently used MAC address is primary and both routes from S3 to S1 are available. As soon as a link on the primary route from S3 to S1 fails, S3's entry becomes <101>, indicating that S3 should use S1's second MAC address to reach S1, and the old primary route is now unavailable.

Conventional Ethernet switches use a source learning mechanism to populate their forwarding table, and thus do not support dynamic routing that could accommodate fluctuating traffic workloads. Only Layer-3 routers provide such support. Most commodity Ethernet switches provide the flexibility to statically and programmatically populate their forwarding table. *Peregrine* leverages this capability to support a centralized routing architecture, in which a route server computes the routes according to a number of optimization criteria, and populates the resulting routes on the switches' forwarding tables. *Peregrine* uses a load-based routing algorithm [7] that dynamically computes routes based on link loads. To support fast fail-over, *Peregrine* extends this algorithm to pre-compute two disjoint routes for each pair of physical servers. To support network QoS, *Peregrine* gives different weights to physical server pairs so that routes computed for higher-priority physical server pairs should travel on less congested network links than lower-priority physical server pairs.

3.4 Fast Fail-Over for Network Failure

The RAS detects a link failure by receiving an SNMP trap about it. Because a switch failure is effectively the same as multiple link failures, when a switch fails, the RAS receives one or multiple SNMP traps indicating

[1] Bit [1:0] indicates the health status of the primary and secondary path. Bit [2] indicates the selected path (0: Primary, 1: Secondary).

failures of links associated with the switch. To ascertain whether such a switch indeed fails, the RAS continuously pings the switch for a period of time (currently set to 1 second) before arriving at a verdict. Therefore, the switch failure detection time is longer than the link failure detection time.

When the RAS detects a link or switch failure, it invokes *Peregrine*'s failure recovery processing algorithm as follows:

1. Given a failed link/switch, RAS consults with the inverse map to identify all physical server pairs whose primary or secondary route traverses through the failed link/switch, and passes these physical server pairs to the DS.
2. For each physical server pair whose primary route is affected, the DS looks up the pair's destination in its address resolution map, turns off the primary route between them in the corresponding address resolution map entry, and notifies the pair's source server to this effect if the pair of servers are actively communicating.
3. For each affected physical server pair, the RAS removes the forwarding table entries associated with its affected primary or secondary route, and computes a new route for it.

Suppose a physical server S1 is affected by a link failure, and there are N_1 other servers that could send packets to S1 over the failed link, but only N_2 of them are actively communicating with S1 at the time of the link failure. So S1's address resolution map entry originally contained a list of N_2 entries to indicate S1's availability status to the N_2 servers *before* the link failure, but the list will grow to N_1 entries *after* the link failure. It is necessary to expand an affected physical server's communication list in its address resolution map entry to correctly instruct those physical servers that are not communicating with the affected server which MAC address to use when they start communicating with the affected server in the future.

Figure 3 illustrates how *Peregrine*'s fast fail-over mechanism works. Initially, VM6's primary and secondary MAC addresses, `mac1` and `mac2`, are prepopulated on the switches along the two disjoint routes by the RAS (step 1). The primary route to VM6 goes through SW2 and SW3 while the secondary route goes through SW1 and SW4. Whenever a link along the primary path from VM3 to VM6 is down, an SNMP trap is sent from the link's adjacent switch to the RAS (step 2), which determines the physical server pairs that are affected by the link failure and passes these affected server pair information to the DS (step 3), which then informs the source of each physical server pair that its associated destination server is reachable only via its secondary

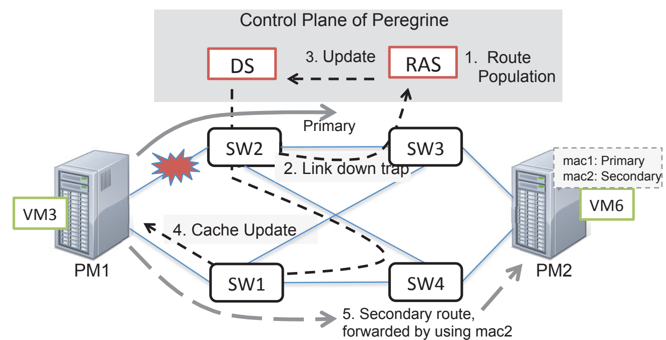

Figure 3: *When a link (e.g. PM1-SW2) fails*, Peregrine *switches every affected server pair (e.g. PM1-PM2) from its primary path (PM1-SW2-SW3-PM2) to its secondary path (PM1-SW1-SW4-PM2).*

MAC address, in this case, sending an ARP entry update to PM1 (step 4) indicating that to send packets from VM3 to VM6 should use `mac2` as the destination MAC address, the secondary MAC address for VM6. After that, all packets destined to VM6 from VM3 will go through VM6's secondary route from this point on (step 5).

3.5 Fast Fail-over for DS/RAS Failure

Because the DS and RAS play a critical role in *Peregrine*'s architecture, it is essential that both of them include a high availability (HA) mechanism to ensure their continued operation despite any single failure of their underlying hardware. First of all, all data structures in the DS and RAS that are required to restart must be kept on disk, and make up their persistent state. We adopt an active master and passive slave architecture, in which the master and slave each have their own local disk. Every update to the master's persistent state is first logged to a memory-resident log, which is synchronously replicated to the slave, and then asynchronously written to the on-disk data structure, which is synchronously replicated to the slave.

The data structures in the RAS that need to be persistent are an in-memory log of pending SNMP traps and the computed routes for every physical server pair and the inverse map that associates network links/switches with routes that traverse them. The route-related information is largely static. The data structure in the DS that needs to be persistent is the address resolution map. *Peregrine* puts RAS's and DS's persistent state in a separate disk volume, and uses DRBD (Distributed Replicated Block Device) [6] to synchronously replicate every write to the master's on-disk persistent state to the slave, and re-synchronize a new slave candidate's on-disk persistent state to the current master's. In addition, *Peregrine* uses *Pacemaker* to monitor the health of the RAS and DS processes and the servers they run on.

The slave takes over as the new master when it detects

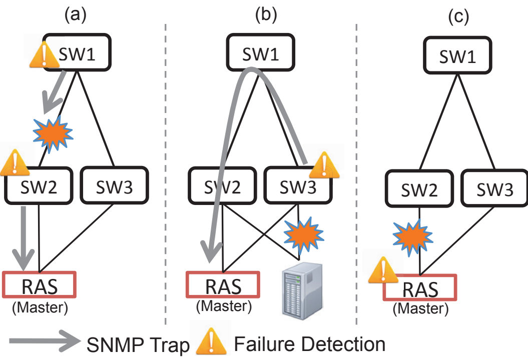

Figure 4: *Because* Peregrine *does not have an out-of-band control network,* Peregrine *must guarantee that SNMP traps be delivered to the RAS despite the failure that triggers the SNMP traps.*

the master dies. When the slave takes over, it recovers the persistent state, performs necessary undoing and redoing, and announces to the world that it is the new master. Specifically, when the slave RAS takes over, it aborts the on-going recovery processing transaction triggered by an SNMP trap if there is one pending, and redoes it from scratch. When the slave DS takes over, it rebuilds the address resolution map from the in-memory log and on-disk copy.

3.6 Resilient Messaging During Fail-over

Because *Peregrine*'s fail-over processing involves failure-detecting switches, the RAS, the DS and the affected physical servers, it is possible that a network failure prevents the communications in its associated fail-over processing and thus inhibits its own recovery. A standard solution to this problem is to install a separate control network for out-of-band fail-over processing. However, such a design is still problematic because there is no guarantee that the out-of-band control network itself won't fail. Instead, *Peregrine* uses in-band signaling to simplify the network infrastructure, but ensures the resiliency of message delivery during fail-over processing by transferring fail-over messages over paths unaffected by the triggering link/switch failures. Fail-over messages include SNMP traps, affected physical server pairs, and notifications to invalidate ARP cache entries.

The HA version of RAS consists of a master node and a slave node, and the HA version of DS also consists of a master node and a slave node. Each RAS/DS node is assigned two MAC addresses and one IP address. Because each RAS/DS node has two MAC addresses, *Peregrine* sets up two disjoint paths between it and every other node that communicates with it, and the DS decides which path should be used between each pair of communicating nodes, including the communications between the two RAS nodes and those between two DS nodes.

Every network switch is configured to send each of its SNMP traps twice, once to the master RAS and the second time to the slave RAS. Figure 4 (a) shows that if the link that fails is between two switches, both switches detect it, and at least one of them is able to send its SNMP traps to the RAS nodes in spite of the failure. If the link that fails is between a switch and a physical server, as in the case of Figure 4 (b), there is only one switch detecting the failure, and this switch definitely is able to send out the SNMP traps associated with this link failure to the RAS. If the link that fails is between a switch and a RAS node, as in Figure 4 (c), this RAS node detects this link failure itself without relying on SNMP traps. Upon receiving an SNMP trap, the master (slave) RAS synchronously replicates it to the slave (master) RAS. This replication serves two purposes: enhancing the reliability of SNMP trap delivery even when the network drops SNMP packets from time to time, and duplicating the in-memory log for RAS fail-over.

The kernel agent on every physical server constantly keeps track of the IP address and the two MAC addresses of the current master DS so that it could submit its ARP queries to the right DS node over a healthy path. In case an ARP query times out, the kernel agent retries the same query with an alternative MAC address. When a new master DS comes along, it broadcasts multiple times to announce to all physical servers its IP address and MAC addresses.

When the master RAS starts up, it establishes a UDP connection with each of the two DS nodes. Through these two UDP connections, the master RAS is able to tell which DS node is the current master DS. When a link/switch failure occurs, the DS is the one that tells every other node whether to switch paths when communicating with specific nodes, except the communication between the RAS and the DS, because this communication takes place *before* the DS is notified of the failure. Therefore, when the master RAS receives an SNMP trap associated with a link/switch failure, it first determines whether it should reach the master DS via its secondary MAC address, and informs the master DS of this failure using the pre-built UDP connection over a path unaffected by the failure. It is crucial that a UDP connection rather than a TCP connection be used here, because the return traffic (e.g. ACK packets) of a TCP connection from the master DS may be blocked by the failure. Once the master DS is informed of a failure, it adjusts its path to the master RAS to bypass the failure if necessary, and then establishes a TCP connection with the master RAS to retrieve the affected physical server pairs.

The master RAS sends the physical server pairs affected by a failure in two batches, the first batch containing those physical server pairs in which the master

DS is the source, and the second batch everything else. The master DS uses the first batch to adjust its paths to physical servers in this batch, and then sends out notifications in the second batch using the adjusted paths. To reduce the messaging overhead of notifications, the master DS further sorts the notifications according to destination nodes, and batches all notifications destined to the same node into as few packets as possible. When the per-server kernel agent receives notifications, it updates its ARP cache and its DS data structure accordingly.

3.7 Broadcast Support

Although *Peregrine* is designed to minimize broadcast traffic, it cannot completely does away with broadcast traffic. For example, ARP requests from network devices on which no *Peregrine* agent is installed, e.g. commercial routers and switches, are broadcast packets. As another example, some applications, e.g. Microsoft Exchange cluster, may use application-level broadcast messages to maintain cluster membership. To accommodate broadcast traffic and prevent Ethernet storms, the RAS sets up a tree that spans all nodes in the entire physical network without using the spanning tree protocol, and allows broadcast packets to flow only on this tree by disabling the broadcast option on all ports that are not on this tree. When a link/switch failure occurs, the RAS amends this tree accordingly to ensure the resulting tree continues to span the entire network.

4 Performance Evaluation

4.1 Evaluation Methodology

We used two racks in the ITRI container computer as the evaluation testbed for the *Peregrine* prototype. The testbed consists of eight 48-port TOR switches each with two 10GE uplink, two 48-port 10GE regional switches, and 88 physical machines. Each physical machine is equipped with eight 2.53GHz Intel Xeon CPU cores, 40GB DRAM, and 4 GE NICs, and is installed with CentOS 5.5, which is equipped with the Linux kernel 2.6.18. Four physical machines are used to deploy the RAS, DS, and their master and slave. The *Peregrine* kernel agent is installed on all other physical machines. Each physical machine is connected to four TOR switches via a separate 1GE NIC, and each TOR switch in turn is connected to four regional switches via a separate 10GE link. No firmware modifications are required on these regional or TOR switches.

To quantify the fail-over latency, we measured the service disruption time for an UDP connection running on two physical machines of the evaluation testbed under various single-failure scenarios. More concretely, the sender of this UDP connection sent one packet every millisecond to the receiver; we then counted the number of lost packets when a failure occurs and *Peregrine*'s fail-over mechanism kicks in, and the resulting number corresponds to the service disruption time.

To assess the efficiency of different fail-over steps, we broke the service disruption time into the following four components:

1. *Failure detection* time: the time between when a failure occurs and when the RAS detects the failure,
2. *Damage assessment* time: the time for the RAS to identify the set of primary and secondary routes affected by a given failure and pass the associated information to the DS,
3. *ARP Update* time: the time for the DS to update its own ARP database entries corresponding to the source nodes of affected physical server pairs and to send out ARP cache updates to the destination nodes of these pairs, and
4. *Switch-over* time: The kernel agent on a physical server updates its ARP cache upon receiving such an ARP cache update message.

To accurately measure the failure detection without installing an agent on the switches, we set up another UDP connection from a source server through the failed link or switch to the RAS, in which the source server also sends a UDP packet to the RAS every millisecond. The RAS measures the time between when it stops receiving packets through this UDP connection (a failure occurs) and when it receives the SNMP associated trap (a failure is detected). Because the *switch-over* time is negligible, we focus on the first three components in the following subsections. Each reported time measurement below is an average of 5 runs.

4.2 Link Failure

Table 1 shows the average service disruption time and its detailed breakdown for four different types of link failures: failure of a link between a server and a 1-GE switch (Server-Switch), failure of a link between a 1-GE switch and a 10-GE switch (Switch-Switch), failure of the link between the DS and a 1-GE switch (DS-Switch), and failure of the link between the RAS and a 1-GE switch (RAS-Switch). The time taken to detect a link failure and send out its associated SNMP trap is much smaller for the 10-GE switches in our testbed, between 60 ms to 80 ms, than for the 1-GE switches, between 200 ms and 1000 ms. We suspect that 10-GE switches detect the link status using event triggering scheme whereas 1-GE switches employ polling-based scheme. In the case of Switch-Switch link failures, it is a 10-GE switch that

Failed Link	No . of Affected Pairs	No. of Notifications	Failure Detection	Damage Assessment	ARP Update	Service Disruption
Server-Switch	158	8	787	13	6	810
Switch-Switch	1383	101	59	88	39	190
DS-Switch	153	73	242	34	30	300
RAS-Switch	156	134	359	29	25	420

Table 1: *The average service disruption times of four different types of link failure and their detailed breakdowns. All time measurements are in terms of ms.*

Failed Switch	No. of Affected Pairs	No. of Notifications	Failure Detection	Damage Assessment	ARP Update	Service Disruption
Regional Switch	6684	203	1881	326	234	1180
Server-Switch	3786	95	1129	156	88	1280
DS/RAS-Switch	6496	343	1407	316	223	1480

Table 2: *The average service disruption times of three different types of switch failures and their detailed breakdowns. All time measurements are in terms of ms.*

sends out the associated SNMP traps, whereas Server-Switch and DS-Switch link failures are detected by 1-GE switches. As for RAS-switch link failures, it is a kernel module in the RAS that detects them directly. As a result, in all cases except Switch-Switch, the failure detection time dominates and accounts for more than 80% of the service disruption time. Unfortunately, the failure detection time is completely determined by the switches and beyond the control of *Peregrine*.

If the failure detection time is excluded, the combined DS and RAS fail-over processing time, which is dictated by *Peregrine*, is below 120 ms for all link failures and below 70 ms if Switch-Switch link failures are ignored. The Switch-Switch link failure entails a much larger number of affected server pairs and notifications than other types of link failures. The damage assessment time is generally proportional to the number of affected server pairs (second column), and the ARP update time is generally proportional to the number of notifications that the DS sends to affected servers (third column). When the RAS sends out the list of server pairs affected by a link failure to the DS, the DS only needs to send ARP updates to destination nodes of a subset of those server pairs that are *actively* communicating with each other at that instant. That is why the number of notifications is smaller than the number of affected server pairs.

4.3 Switch Failure

Table 2 shows the average service disruption time and its detailed breakdown of three different types of switch failures: failure of a 10-GE regional switch (Regional Switch), failure of a 1-GE switch connected to a physical server (Server-Switch), and failure of the switch connected to both RAS and DS (DS/RAS-Switch). The switch failure detection time is generally higher than the link failure detection time because the RAS needs to receive multiple SNMP traps associated with link failures of a suspect switch and ping the suspect switch for 1 second without getting any response before concluding that the switch fails. The failure detection time for Regional Switch is higher than that for Server-Switch because the former is detected by 1-GE switches whereas the latter is detected by 10-GE switches. The switch failure detection time for the DS/RAS-Switch failure is about 1-second ping delay plus the link failure detection time for the RS-Switch failure, because both are detected by RAS.

For the Server-Switch and DS/RAS-Switch failure, the service disruption time for a switch failure is smaller than the sum of failure detection time, damage assessment time and ARP update time because a portion of fail-over processing is triggered by link failure SNMP traps and is thus overlapped with the switch failure detection time. The fail-over processing for those link failures whose SNMP traps cannot be successfully delivered to the RAS is triggered only after the RAS concludes that a switch failure occurs. The extent of overlap for the DS/RAS-Switch failure is higher than that for the Server-Switch failure because a significant portion of a DS/RAS-Switch failure's fail-over processing is due to the fail-over processing of the DS-Switch and RAS-Switch link failures and they are completed before the DS/RAS-Switch failure is detected. In the case of the Regional Switch failure, the service disruption time is actually smaller than the failure detection time because the fail-over processing for all the constituent link failures of a switch failure is completed before the RAS concludes that the switch failure indeed takes place.

4.4 RAS and DS Failure

When the master RAS fails, it takes on average 1038 ms for the RAS slave to notice because the RAS slave probes the RAS master for 1000 ms before declaring a take-over, and another 0.45 ms to restart itself. The restart processing of the RAS slave is fast because the only RAS persistent state is the pending SNMP trap log, which is mostly empty in this test. Because the RAS performs fail-over processing for link/switch failures, failure of the RAS master potentially increases the fail-over

Address Resolution Map Size	Service Disruption Time	Failure Detection Time	DRBD Switch Time	DS Recovery Time
6556 Entries	2811	1871	704	269
32469 Entries	3282	1882	706	736

Table 3: *The service disruption time of the DS because of a DS failure and its breakdown under two testbed sizes. All time measurements are in terms of ms.*

latency of link/switch failures. To test this, we turned off the RAS master, then immediately turned off the switch to which the RAS is connected, and measured the service disruption time of a UDP connection going through the switch. The service disruption time is increased to 1580 ms, which, as expected, is about 1000 ms higher than the average fail-over latency for link failures shown in the last subsection.

Table 3 shows the service disruption time of the DS; a DS failure is about 2811 ms and 3282 ms when the address resolution map contains 6556 entries and 32469 entries. We used a special test program that continuously submits ARP queries every 50 ms to the master DS and slave DS, and the service disruption time corresponds to the time interval between when the master DS stops responding and when the slave DS starts responding. The DRBD switch time corresponds to the time the slave DRBD needs to mount the replicated disk partition before becoming the new master DRBD. The failure detection time is bound by the Pacemaker library used in the current *Peregrine* prototype. Both the DS failure detection time and the DRBD switch time remain unchanged as the testbed size is increased. In contrast, the DS recovery time is proportional to the size of the address resolution map, because larger address resolution maps require longer reload time during recovery.

5 Conclusion

Peregrine is a software-defined network that uses commercial off-the-shelf Ethernet switches as basic building blocks and was originally designed for the ITRI container computer. It uses a centralized control plane to program the forwarding tables and configure the options of the switches in the network. Compared with conventional Ethernet architecture, *Peregrine* is more scalable because it supports dynamic load-based routing, and is more available because it provides self-adaptive fault tolerance against any single failure. More concretely, through proactive primary/secondary routing, *Peregrine* is able to significantly cut down the service disruption time due to link failures, switch failures and control plane failures. The specific research contributions of this work include

- A proactive disjoint path-based primary/secondary routing scheme that is able to quickly switch communications between server pairs affected by a link/switch failure to their pre-arranged alternative routes,
- A highly available control plane that is capable of continued operation despite any single control server failure,
- A resilient communication design that achieves reliable message transfer in fail-over processing of a link/switch failure without using any out-of-band control network, and
- A fully operational prototype that is able to cut down the service disruption time associated with any single link failure to under 120 ms, if the failure detection time is excluded.

References

[1] AWERBUCH, B., ET AL. Distributed control for paris. In *Proc. ACM PODC 2012*.

[2] BEHESHTI, N., AND ZHANG, Y. Fast failover for control traffic in software-defined networks.

[3] BOTTORFF, P., AND HADDOCK, S. Ieee 802.1 ah-provider backbone bridges, 2007.

[4] CASADO, M., ET AL. Ethane: Taking control of the enterprise. *Proc. ACM SIGCOMM 2007*.

[5] CHIUEH, T., ET AL. Peregrine: An all-layer-2 container computer network. In *Proc. IEEE Cloud, 2012*.

[6] ELLENBERG, L. Drbd 9 and device-mapper: Linux block level storage replication. In *Proc. of the 15th International Linux System Technology Conference, 2008*.

[7] GOPALAN, K., ET AL. Load balancing routing with bandwidth-delay guarantees. *Communications Magazine, IEEE, 2004*.

[8] GREENBERG, A., ET AL. A clean slate 4d approach to network control and management. *Proc. ACM SIGCOM, 2005*.

[9] GREENBERG, A., ET AL. Vl2: A scalable and flexible data center network. *Proc. ACM SIGCOMM 2009*.

[10] IANNACCONE, G., ET AL. Analysis of link failures in an ip backbone. In *Proc. ACM SIGCOMM 2002*.

[11] KOMPELLA, K., ET AL. Link bundling in mpls traffic engineering (te).

[12] KOPONEN, T., ET AL. Onix: A distributed control platform for large-scale production networks. *Proc. USENIX OSDI 2010*.

[13] MARKOPOULOU, A., ET AL. Characterization of failures in an ip backbone. In *Proc. IEEE INFOCOM 2004*.

[14] MCKEOWN, N., ET AL. Openflow: enabling innovation in campus networks. *ACM SIGCOMM 2008*.

[15] NIRANJAN MYSORE, R., ET AL. Portland: a scalable fault-tolerant layer 2 data center network fabric. *Proc. ACM SIGCOMM, 2009*.

[16] RODEHEFFER, T. L., AND SCHROEDER, M. D. *Automatic reconfiguration in Autonet*. ACM, 1991.

[17] SHARAFAT, A. R., ET AL. Mpls-te and mpls vpns with openflow. In *Proc. ACM SIGCOMM 2011*.

[18] SHERWOOD, R., ET AL. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep* (2009).

[19] STAESSENS, D., ET AL. Software defined networking: Meeting carrier grade requirements. In *Proc. IEEE LANMAN, 2011*.

[20] TAVAKOLI, A., CASADO, M., KOPONEN, T., AND SHENKER, S. Applying nox to the datacenter. *Proc. HotNets 2009*.

[21] ZHANG, Y., ET AL. On resilience of split-architecture networks. In *IEEE GLOBECOM 2011*.