February 2012

z/VM Single System Image Cluster Live Guest Relocation with HiperSockets: Exploitation via Dynamic Routing

R. Brenneman, G. Markos, E. M. Dow
{rjbrenn, markos, emdow}@us.ibm.com
IBM Platform Evaluation
Test Laboratory
Poughkeepsie, NY

Table of Contents

1 Problem Statement Abstract

This paper describes a solution that allows customers running Linux® on z/VM® to exploit the new z/VM Live Guest Relocation (LGR) functionality offered in the new z/VM 6.2 release while still making use of the IBM zEnterprise™ System HiperSockets™ technology as Linux guests are relocated around the z/VM Single System Image cluster. This solution makes use of IBM's Operations Manager product for z/VM to automate the network configuration changes that enable Linux to use HiperSockets for whichever CPC it is running on, while the use of dynamic routing enables Linux to intelligently exploit the appropriate HiperSockets interface without application configuration changes.

In 2010 and early 2011 we put significant effort into configuring our OSPF dynamic routing environment so that we were able to automatically use HiperSockets connections to reach endpoints within the local CPC, and also use real OSA devices or z/VM VSWITCH connections to reach endpoints outside the local CPC. We also started working with early test releases of z/VM 6.2 during this same time, and realized that the new Live Guest Relocation function would seriously impact our ability to use HiperSockets automatically based on OSPF dynamic routing.

More specifically: each HiperSockets must appear as a unique network within the OSPF area. Moving a virtual machine from one CPC to another will remove that virtual machine from one HiperSockets network segment and adds it to another. Live Guest Relocation requires equivalent network connectivity on both the source and target systems. As one can infer, guest relocation and HiperSockets seem intrinsically at odds when one considers the requirement that each HiperSocket be a unique network segment in the OSPF architecture.

After gaining experience with z/VM 6.2 Live Guest Relocation and its restrictions and requirements, we were able to design a solution to address the issues that arise when attempting to use these three technologies (OSPF, HiperSockets, and LGR) together.

2 Solution’s Components - Discussion

Our system environment consists of four CPCs: IBM System z10 Business Class™ (z10 BC™), IBM System z10® Enterprise Class (z10 EC™), IBM zEnterprise 114 (z114) and a IBM zEnterprise 196 (z196). All four CPCs are in use by both the z/OS Integration Test team and our Linux Virtual Server Integration Test team. The z/OS® team has a set of LPARs on each CPC that hosts their two Parallel Sysplex® clusters, and the LVS team has a separate set of LPARs for z/VM and Linux systems. The z/VM systems had been clustered together in previous releases of z/VM (up to release 6.1) using the Cluster Systems Extensions (CSE) technology. CSE enabled us to manage multiple z/VM systems as a group, and provided a limited ability to move virtual machines between LPARs in order to adjust for varying workload requirements. Moving virtual machines between z/VM LPARs on different CPCs required network changes to enable the Linux system to exploit the new HiperSockets connectivity on the new CPC.

z/VM 6.2 introduced Single System Image (SSI) style clustering, which in turn enables Live Guest Relocation (LGR). We migrated our z/VM 6.1 systems, which used CSE clustering, up to z/VM 6.2 with SSI clustering. Our SSI migration experiences are documented in another paper called “Early Experiences with z/VM 6.2 and Live Guest Relocation: moving from CSE to SSI” which is located at: **ibm.com**/systems/services/platformtest/servers/systemz_library.html#related_publications.

After we completed the SSI migration we realized that we needed to design a solution that would continue to provide high availability without requiring any manual configuration changes, or without requiring changes to any of the components within the underlying infrastructure. The solution would need to seamlessly support our OSPF configuration as guests with HiperSockets interfaces were actively relocated across different CPCs. We chose to use the Operations Manager for z/VM product to automate a portion of this process. We discuss each of the major components of the solution (SSI Clustering, LGR, and Operations Manager) in detail in the following sections.

2.1 SSI Clustering

z/VM 6.2 introduced a new clustering technology called Single System Image (SSI). SSI enables a cluster of up to four z/VM systems to be managed as if they were a single system. The members of the SSI cluster share a common z/VM Directory, share common spool space, and are able to use basic VM system commands across cluster members.

SSI provides the ability to have a common z/VM system directory by making some changes to the structure of the directory itself. A *guest user* is defined normally as in previous versions of z/VM and it exists on every member of the SSI cluster, but is only permitted to log on to one member at a time. A new construct called an *identity user* was added to the SSI directory structures. An identity user is similar to a guest user, except that it is not required to be identical on every SSI cluster member, and it can log on to multiple cluster members simultaneously. Essentially a guest user is used to run the actual business applications and an identity user is used to manage the z/VM system and provide common services such as TCP/IP connectivity and RACF® protection.

z/VM 6.2 also provides a common view of the contents of Spool space for guest users. As a guest user moves around to different SSI cluster members by either LGR or logging off and on, they see the same contents in their virtual readers and virtual printer queues. This provides a seamless view of the SSI cluster as if it were a single system. Identity users are not presented with this single unified view of spool space. They see only the spool files that are local to whichever SSI member they happen to be logged on to at the moment.

Lastly, z/VM 6.2 extends some z/VM system commands and messaging functions across the SSI cluster members so that the cluster may be managed from a single member.

2.2 Live Guest Relocation

In a z/VM SSI configuration guest users, but not identity users can be relocated from one cluster member to another while they are running, without interrupting the operation of the guest operating system. The guest operating system might not even notice that a guest user has been relocated.

LGR comes with a lengthy list of restrictions on the guest user configurations which can be relocated. A Linux guest that only uses virtual resources (mdisk, VSWITCH) and does not rely on any resources local to a specific SSI cluster member, is usually eligible for relocation. When you add HiperSockets devices to a Linux guest, equivalent HiperSockets devices must be available on both the source and target SSI cluster members.

Therein lies the difficulty. z/VM's definition of *equivalent networks* means that a host connected to a network must be directly reachable to the exact same network through equivalent network devices on each z/VM SSI cluster member. In networking terms, *equivalent network devices* mean that the guest must have a directly attached interface for the same network segment, regardless of which z/VM SSI cluster member the guest is located on.

Because HiperSockets networks are physically bound within the limits of a given CPC, a HiperSockets network on one CPC cannot possibly be directly attached to guests located on a different CPC. Furthermore, if two HiperSockets, each being located on separate CPCs, were defined on the same network segment, we would have a disjoint network where half the hosts would not be able to talk to the other half because they would be split across two CPCs. For a visual representation of these points, refer to Figure 1: Duplicate HiperSockets subnets across multiple CPCs.

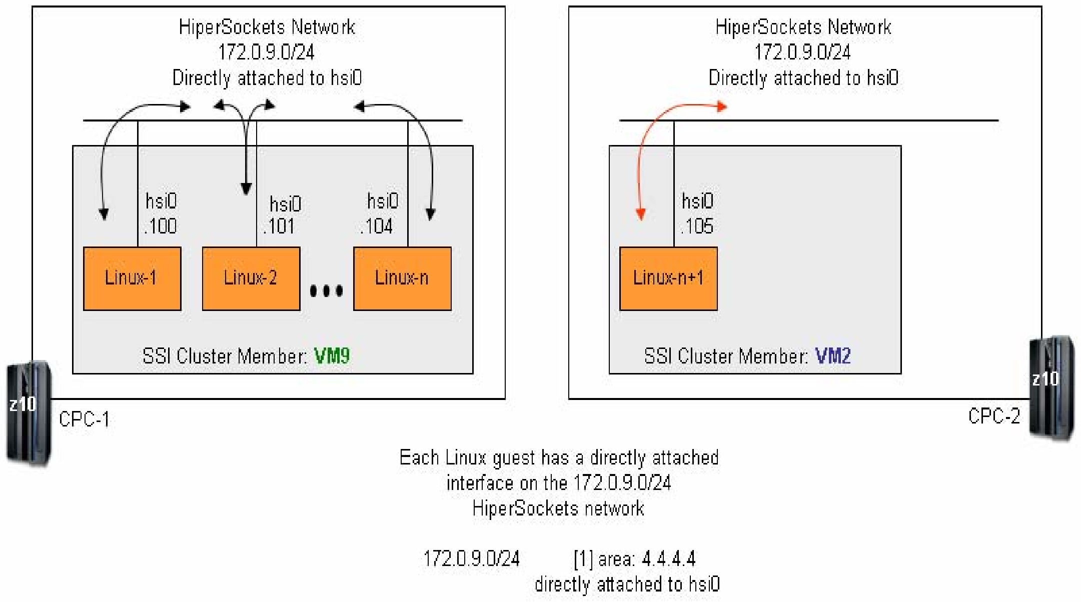


Figure 1: ***Duplicate HiperSockets subnets across multiple CPCs***

2.3 Operations Manager

Operations Manager for z/VM is an automation product that manages console message traffic for z/VM systems and their guests. Each guest virtual machine is configured such that their console messages are sent to the Operations Manager system for processing. Operations Manager scans the inbound console messages from the guests and matches the messages against a set of user-defined rules. Operations Manager can then automatically run scripts based on which rules match, and thus can automate a large portion of the day-to-day operations of a z/VM system.

We used this ability in Operations Manager to run a script specifically designed to dynamically handle making the required network changes to our guests with HiperSockets to ensure that OSPF would behave correctly within the context of our LGR enabled z/VM SSI cluster.

Additional information for Operations Manager for z/VM can be found at:
ibm.com/software/sysmgmt/zvm/operations/library.html

3 Prior Work

IBM mainframes have a long published history when it comes to providing High Availability (HA). While it is not this paper's intention to provide details on previously established HA best practices, this paper does leverage established best practices when possible.

Specifically, the HA best practices presented in the following papers, were utilized in this work:
ibm.com/systems/services/platformtest/servers/systemz_library.html#related_publications,

1. "A Reference Implementation Architecture for Deploying a Highly-Available Networking Infrastructure for Cloud Computing and Virtual Environments using OSPF"
2. "Validation of OSPF on IBM Linux on System z at Scale"

   In addition to these two white papers, the following Share presentation, "Dynamic Routing: Exploiting HiperSockets and Real Network Devices", provides a discussion on utilizing HiperSockets within a dynamically routed infrastructure. This presentation can be downloaded from: http://share.confex.com/share/117/webprogram/Handout/Session9477/OSPF_Linux_Z.pdf.

4 High Availability

In high availability configurations, it is important to always present end users with a consistent IP address for a given service, regardless of changes to the underlying network infrastructure used to deliver that service. Virtual IP Addressing (VIPA), HiperSockets, Open Shortest Path First (OSPF) dynamic routing protocol, Open System Adapter (OSA), Linux for IBM System z®, z/VM, and z/VM Virtual Switch (VSWITCH) are all combined to provide the high availability infrastructure needed to support this paper's solution.

4.1 Virtual IP Addressing (VIPA)

A traditional IP address is associated with each end of a physical network link. Within an IP routed network, failure of any intermediate link, or failure of a physical network adapter will disrupt end user service unless there is an alternate path available through the routing network.

A Virtual IP Address (VIPA) removes the physical network adapter as a single point of failure by associating an IP address with the server's TCP/IP stack (by associating it with a "dummy" device), subsequently making the VIPA accessible across all of the stack's physical interfaces and not limiting it to a specific physical network attachment. Therefore, because a VIPA does not have a physical network attachment associated with it, a VIPA is active as soon as the TCP/IP stack is active and it becomes accessible across each of the stack's respective physical interfaces as each interface becomes active.

To the routed network, a VIPA appears to be a host destination (a 32-bit network prefix for IPv4) on a multi-homed TCP/IP stack. When a packet with a VIPA destination reaches the TCP/IP stack that owns that particular VIPA, the IP layer recognizes the address as an address in the TCP/IP stack's Home list, passing it up to the transport protocol layer in the stack for application processing. Note that a VIPA should be on a different subnet than other IP addresses defined for that host. External routers must then be told that they can reach that VIPA through IP addresses associated with each of the network adapters on that host.

4.1.1 ARP_ANNOUNCE

We found that a specific Linux sysctl configuration flag needed to be set to support our High Availability environment to ensue that the Linux guest's VIPA would not be announced, or used in any ARP requests. In other words, we needed to set each of the guest's physical interfaces to use only its local address in all ARP request exchanges and not use any address available to the interface, such as the VIPA, in any ARP exchanges.

This task can be accomplished by setting the “arp_announce” flag to either a 1 or, 2 on an interface-by-interface basis, or globally across all interfaces. We chose to set this flag globally by setting the following flag to a 1 on each of our guests. The value of this global setting can be read or set by reading or writing to the following proc file:

```
/proc/sys/net/ipv4/conf/all/arp_announce
```

Note: The precise method to persistently set this value varies across Linux distributions and is left as an exercise for the reader. Changes made by echoing values into proc at run time will not persist across Linux system reboots.

4.2 HiperSockets

HiperSockets is a Licensed Internal Code (LIC) function that emulates the Logical Link Control (LLC) layer of an OSA-Express QDIO interface to provide very fast TCP/IP communications between z/OS and Linux servers running in different LPARs and z/VM guests within a single System z server.

HiperSockets uses internal Queued Input/Output (iQDIO) protocol, setting up I/O queues in the System z processor’s memory so that traffic passes between virtual servers at memory speeds, totally eliminating the I/O subsystem overhead and external network delays.

The virtual servers that are connected to HiperSockets form a *virtual LAN*. Multiple independent HiperSockets virtual LANs are supported within a single System z processor, as well.

Adding a HiperSockets can provide an inexpensive way to make each Linux guest multi-homed, which is a fundamental tenet of serious HA deployments. By providing server connectivity to more than one broadcast domain, dynamic routing can be leveraged to ensure that packets can be redirected around network failures.

4.3 Open Shortest Path First (OSPF)

While VSWITCH and HiperSockets provide acceptable networking redundancy, redundancy in itself is not always sufficient to provide a satisfactory level of high availability. Dynamic routing protocols, such as Open Shortest Path First (OSPF), enable networks to take advantage of redundant connectivity to route traffic around network failures, as well as across newly added links. OSPF by itself can provide an additional level of availability to any solution, however utilizing Virtual IP Addressing as part of the OSPF configuration will further increase a solution’s availability.

Multi-homed servers (those servers with multiple NICs) combined with a VIPA removes a solution’s dependency on a physical network adapter by associating the VIPA with the server’s TCP/IP stack so that the address is accessible via all of the server’s active network interfaces. Should one interface experience a link failure, either planned, or unplanned, the OSPF routing protocol will dynamically recognize that a route to that same host is still available over one of the server’s other remaining active interfaces.

Copies of our OSPFD and Zebra configurations can be found in sections 6.3 and 6.4 on page 39.

4.4 OSPF and HiperSockets with SSI LGR

Preserving network connectivity while relocating Linux guests from one z/VM host to another z/VM host is transparent for Linux guests that do not have any HiperSockets interfaces however, additional consideration is required for relocating guests that do have HiperSockets interfaces.

Because HiperSockets exist only in System z processor's memory and therefore, HiperSockets networks on different CPCs are physically disjoint. Each HiperSockets network within a dynamically routed topology must be unique across CPCs, otherwise routing loops would be introduced. Therefore, in order to relocate a guest with HiperSockets from one CPC to another, a method is needed to make the guest *CPC aware* to ensure that the correct HiperSockets interface is always online with respect to the guest's CPC residency.

Our approach to ensuring that a guest has the correct HiperSockets interface online with respect to its CPC residency involved utilizing a combination of REXX™ execs and a Bash script that will be discussed in section 4.6.

4.5 The challenge and how we met it

As was pointed out in section 2.2 Live Guest Relocation on page 4, z/VM requires that a guest have network equivalency on both the source and target SSI cluster members in order for the guest to be eligible for relocation. However, because each CPC's HiperSockets interface must be on a unique network within a dynamically routed infrastructure, SSI Clustering, Operations Manager, and LGR add new challenges in preserving HA for guests with HiperSockets as they are relocated across SSI Cluster members.

Specifically, in order for a guest with a HiperSockets interface to be eligible for relocation between multiple CPCs, it must actually have multiple HiperSockets interfaces defined, with each interface configured to join a different HiperSockets subnet. However, safeguards must also be in place to ensure that the guest has one, and only one, of those HiperSockets interfaces up (on-line) at any point in time, with respect to the specific SSI cluster member that the guest currently resides on.

To overcome this challenge we utilized a combination of REXX and Bash scripts that were designed to provide the guest with a level of CPC awareness to ensure that the guest will always bring up the correct HiperSockets interface depending on which SSI cluster member the HiperSockets guest resides on.

4.6 REXX and Bash Automation

We created two REXX EXECs and one Bash script to handle the Live Guest Relocations for our guests that utilize HiperSockets. The first REXX EXEC expects as input a workload parameter that identifies which group of guests is to be relocated via the Live Guest Relocation feature. This first EXEC validates the workload parameter, then calls a second REXX EXEC to perform the actual Live Guest Relocations.

The second REXX EXEC determines if a guest being relocated has HiperSockets interfaces, If so, the second REXX EXEC starts a Bash script running on the respective Linux guest virtual machine that configures the correct HiperSockets interface up (with respect to the guest's final CPC residency). Copies of these REXX EXECs can be found in the Appendix; section 6.1 REXX EXECs.

4.6.1 Bash Script Automation

As we mentioned earlier, in order to relocate a guest across SSI Cluster members, z/VM requires that the guest have equivalent network devices on each member that the guest can be relocated to. Because all of our guests, including our HiperSockets guests, have equivalent VSWITCH connections across all members in the SSI cluster, relocating guests without HiperSockets interfaces is transparent. Therefore, we designed a Bash script that was run only on our HiperSockets guests to provide the guest with a level of CPC awareness, as well as to provide timely actions that precipitate specific network routing updates across the OSPF area. A copy of the Bash script can be found in the Appendix; section 6.2, Bash Script.

With those basic premises in mind, our Bash script's design points were relatively simple. We needed to make sure that no traffic would be directed over a guest's HiperSockets interface while the respective guest was being relocated. In other words, we needed the HiperSockets guest to send an OSPF link state advertisement to indicate that any routes across the HiperSockets network to its respective VIPA were no longer active, and therefore, all traffic to its VIPA needed to dynamically be redirected across one if its two data center interfaces, named trk1 or trk2. Our Bash script accomplished this by unconditionally configuring the guest's HiperSockets interfaces down (off line) immediately prior to the guest being relocated. This forced the guest's OSPF process to immediately send out an OSPF link state advertisement indicating that the respective HiperSockets link was no longer active on the guest.

After a guest's HiperSockets interface was configured down, the guest would then appear as if it were a non-HiperSockets guest and, as we mentioned earlier, relocation of a non-HiperSockets guest is transparent due to the VSWITCH networking equivalency inherent in our network design.

The last task that our Bash script needed to perform was two-fold. First it needed to determine which member in the SSI cluster the guest was relocated to, or in the case of a failed relocation, which member in the SSI cluster that the guest currently resides on so that it can activate the correct HiperSockets interface. Second, this action implicitly forces the OSPF process to send an OSPF link state advertisement, but this time the link state advertises that a new route across the respective CPC's HiperSockets network is now available to reach the guest's VIPA.

4.6.2 Linux Console (MINGETTY) Auto logon

In order for our REXX Exec to be able to start the Bash script on the Linux guest, there must be an active console session running on the guest. Also, after a guest is relocated the console device must automatically be restarted. To accomplish these tasks we utilized the mingetty auto logon method that will automatically log a user ID onto the guest's console device at Linux boot time and will re-establish, or re-drive the console device when the guest has been relocated.

For details on the mingetty auto logon method, refer to the publication at:
ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101634

4.6.2.1 SUDOERS

Our Bash script needed to issue a few privileged commands on the guest's console, which we were able to authorize by placing the following statements into the /etc/sudoers file.

```
# Cmnd alias specification
Cmnd_Alias LXCONUSR_CMDS = /sbin/modprobe, /sbin/vmcp, /sbin/ifup, /sbin/ifdown

# User privilege specification
lxconusr ALL=(ALL) NOPASSWD:LXCONUSR_CMDS
```

5 Experimental Evaluation

It is beyond the scope of this paper to discuss high availability and best practices for the Live Guest Relocation feature for guests with HiperSockets interfaces that are located within flat, or statically routed networking infrastructures.

Therefore, this section of the paper begins by describing our dynamically routed network topology, followed by citing the key availability metrics captured in a baseline scenario. We then contrast these baseline metrics with metrics obtained as we systematically relocate Linux guests from one z/VM host to another z/VM host demonstrating how the Live Guest Relocation feature, in conjunction with OSPF, provides high availability.

5.1 Network Topology Diagram and Key Network Design Points

The following graphic, Figure 2: Dynamically Routed Network Topology along with Table 1: Linux Host IP Assignments provide visual representations of our networking topology. The key design points to understand in this figure and table are as follows:

- There are two z/VM hosts (VM2 and VM9) with each residing on different System z10 CPC.
- Each Linux guest's VIPA is assigned to the guest's dummy0 interface, it is a 32-bit host address (10.20.7.nn/32), and is the address used for each guest in our Domain Name Servers (DNS).
- Each of our System z10 CPCs have HiperSockets configured.
- Each HiperSockets network is in a different subnet, because HiperSockets cannot span CPCs, as was illustrated in Figure 1: Duplicate HiperSockets subnets across multiple CPCs on page 5.
- Not all of our Linux guests have a HiperSockets interface
- Each Linux guest is configured with a pair of fully redundant VSWITCHes, such that each VSWITCH connects to a different subnet (10.20.8.0/24 or 10.20.9.0/24) in our data center's OSPF Stub area.
- Not every guest has HiperSockets interfaces. However, the Linux guests that do have HiperSockets interfaces are configured with a pair of fully redundant HiperSockets interfaces, such that each HiperSockets interface connects to a different subnet, but only one HiperSockets interface is active at a time, depending on which SSI Cluster member the guest currently resides on.
  - Note: a solid line indicates the active, on-line, HiperSockets interface for the respective CPC in the diagram.
  - Conversely, a dashed, grayed out line indicates the inactive, off-line, HiperSockets interface for the respective CPC in the diagram.
- All VSWITCHes, HiperSockets, and VIPAs are in the same OSPF Stub area (4.4.4.4)

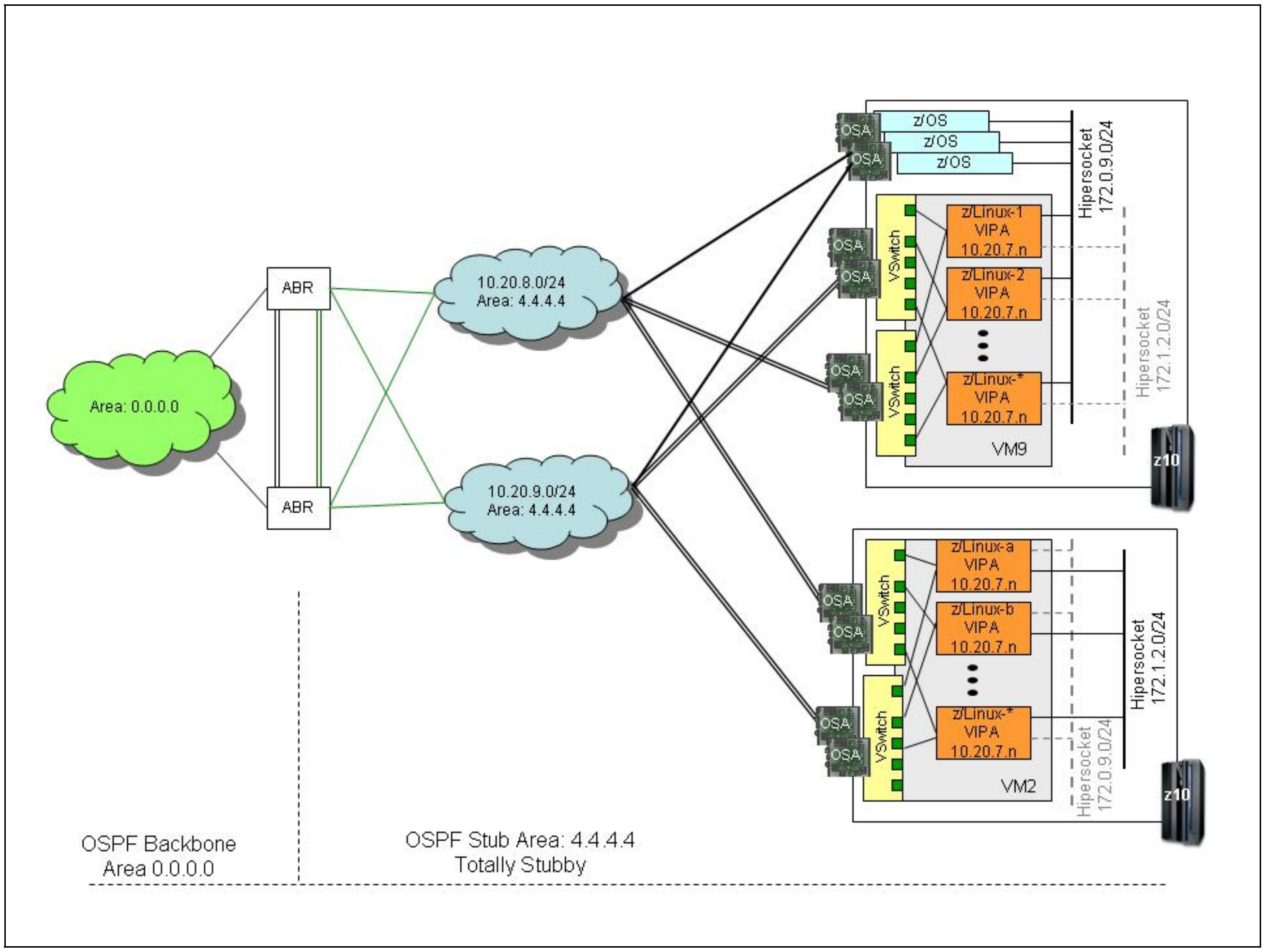


Figure 2: ***Dynamically Routed Network Topology***

Notes:

- A solid line with respect to the HiperSockets subnets indicates the active, on-line HiperSockets interface for the respective CPC in this diagram.
- Conversely, a dashed, grayed out line indicates the inactive, off-line HiperSockets interface for the respective CPC in this diagram.

VIPA and HiperSockets Interface IP Address Assignments				
Type of Guest	Linux Guest Host Name	VIPA subnet/mask	Desired Active Hipersockets Interface with Respect to SSI Cluster Residency	
			VM9 hsi0 Interface subnet/mask	VM2 hsi1 Interface subnet/mask
Without HiperSockets Interfaces	ECMSS00	10.20.7.69/32	n/a	n/a
	LITSHA22	10.20.7.199/32	n/a	n/a
With HiperSockets Interfaces	LITDCON1	10.20.7.100/32	172.0.9.100/24	172.1.9.100/24
	LITSWAS1	10.,20.7.101/32	172.0.9.101/24	172.1.9.101/24
	LITDAT01	10.20.7.104/32	172.0.9.104/24	172.1.9.104/24

Table 1: ***Linux Host IP Assignments***

5.2 Evaluation Discussion

This section of the paper compares availability metrics, in the form of OSPF routing table displays, from each of our five Linux guests.

We prefer to use the OSPF routing tables in our assessments, rather than Linux kernel's routing tables. This is because the OSPF kernel supports Equal Cost Multiple Pathing (ECMP) and therefore, the OSPF routing tables include multiple equal cost routes. The Linux kernel's routing table includes only one route, regardless of whether multiple routes are available. Note that for brevity, we only include the pertinent routes between our five guests in our comparisons.

The baseline scenario begins with all five of our Linux guests located on the same z/VM host (VM9). The hostnames for these guests are ECMSS00, LITDCON1, LITSWAS1, LITDAT01, and LITSHA22. Of these five guests, LITSHA22 and ECMSS00 do not have any HiperSockets interfaces while the LITDCON1, LITSWAS1, and LITDAT01 guests have HiperSockets interfaces.

From the OSPF routing table comparisons we observe:

1. For Linux guests located on the same CPC:
    a) For the guests that have a HiperSockets interface, the preferred route to a neighboring guest's VIPA is via the HiperSockets network.
    b) For the guests that do not have a HiperSockets interface, the preferred equal cost routes to a neighboring guest's VIPA is via either one of the guest's two VSWITCH interfaces (named trk1 or trk2)
2. For Linux guests located on different CPCs, the preferred equal cost routes between each guest's VIPA is via either one of the two VSWITCH interfaces (named trk1 or trk2), regardless of whether each guest has a HiperSockets interface on its respective CPC, or not.

5.3 Baseline Evaluations

We begin by comparing the pertinent OSPF routing entries for all five Linux guests. These comparisons are intended to demonstrate that network connectivity for guests without HiperSockets remains constant, regardless of which SSI Cluster member (z/VM host) the non-HiperSockets guest is either located on, or relocated to.

Conversely, these comparisons also show how and why special attention is needed when utilizing the Live Guest Relocation feature to relocate Linux guests with HiperSockets interfaces across SSI Cluster members.

The following table lists the SSI cluster member (VM host) and the z/VM host residency of our Linux guests for the Baseline scenario.

Baseline Residency	Linux Guest	HiperSockets Interface			VIPA
		y/n	hsi0	hsi1	
VM2					
VM9	ECMSS00	no	---	---	10.20.7.69
	LITDCON1	yes	172.0.9.100	---	10.20.7.100
	LITSWAS1	yes	172.0.9.101	---	10.20.7.101
	LITDAT01	yes	172.0.9.104	---	10.20.7.104
	LITSHA22	no	---	---	10.20.7.199

5.3.1 Baseline Evaluation: Common routing entries

The following routing entries are common across all of our guests. Therefore, in the interest of brevity, we will discuss them here and will point out the unique routing entries as they appear in each respective guest's evaluation section that follows.

1) The ABR injects two equal cost Intra Area (IA) default routes (0.0.0.0/0) into each guest's routing table across each guest's directly connected data center network interfaces, named trk1 and trk2
2) Each guest's VIPA, 10.20.7.nnn/32, is assigned to the guest's dummy0 interface, as expected.
3) Each guest has interfaces (named trk1 and trk2) directly connected into each of the two networks located in the data center's 4.4.4.4 OSPF Stub Area.
4) The ABR, router id 10.20.0.1, is connected as the .1 host on both the 10.20.8.0/24 and the 10.20.9.0/24 data center OSPF networks.

```
============ OSPF network routing table ============
N IA 0.0.0.0/0            [20] area: 4.4.4.4                 see note 1)
                          via 10.20.8.1, trk1
                          via 10.20.9.1, trk2
N    10.20.8.0/24         [10] area: 4.4.4.4                  see note 3)
                          directly attached to trk1
N    10.20.9.0/24         [10] area: 4.4.4.4                  see note 3)
                          directly attached to trk2
N    10.20.7.nnn/32       [1] area: 4.4.4.4                   see note 2)
                          directly attached to dummy0
============ OSPF router routing table =============           see note 4)
R    10.20.0.1            [10] area: 4.4.4.4, ABR
                          via 10.20.8.1, trk1
                          via 10.20.9.1, trk2
```

5.3.2 Baseline Evaluation: HiperSockets Guest Routing Table Entries

While the following OSPF routing table excerpt was obtained from one guest with a HiperSockets interface, it is virtually identical to the routing table entries received on all three of our guests with HiperSockets interfaces. Therefore, we will use this one table to represent the routes seen from all three of our guests with HiperSockets interfaces.

The routing entries used in this section were acquired from the LITDCON1 guest that does have a HiperSockets interface. Just as in the previous baseline scenario, all five of our guests are still located on the same VM9 SSI Cluster member for this scenario.

In addition to the common routing table entries previously noted in section 5.3.1, Baseline Evaluation: Common routing entries on page 13, this routing table's entries show the following:

1) This guest has a directly attached HiperSockets interface named hsi0 on the 172.0.9.0/24 HiperSockets subnet.
2) There are two other guests, LITSWAS1 and LITDAT01 at 10.20.7.101 and 10.20.7.104, respectively that are also directly attached to this CPC's HiperSockets network.

While not obvious from this sample capture, we can also conclude that there are no guests with HiperSockets Interfaces located on another CPC, otherwise we would also see indirect network routing entries to reach another HiperSockets subnet.

This guest was chosen to demonstrate two points. First, the shortest path to another guest's VIPA for other guests that have a HiperSockets interface and that reside on the same SSI Cluster member, will be via the directly connected HiperSockets interface, as was cited in section 5.2 Evaluation Discussion, bullet 1. a), on page 12.

- Refer to the 10.20.7.101/32 and 10.20.7.104/32 entries in the routing table below

The second point that we would like to emphasize at this time is that the shortest path to a VIPA on a guest that does not have a HiperSockets interface, such as LITSHA22 and ECMSS00 at 10.20.7.199 and 10.20.7.69 respectively, will always be via the guest's directly connected data center connections (named trk1 and trk2), as was previously mentioned in section 5.2 Evaluation Discussion, bullet 1. b), on page 12.

- Refer to the 10.20.7.199/32 and the 10.20.7.69/32 entries in the routing table below

```
litdcon1> sho ip ospf route
============ OSPF network routing table ============
N    10.20.7.69/32         [11] area: 4.4.4.4
                           via 10.20.8.69, trk1
                           via 10.20.9.69, trk2
N    10.20.7.100/32        [1] area: 4.4.4.4
                           directly attached to dummy0
N    10.20.7.101/32        [2] area: 4.4.4.4
                           via 172.0.9.101, hsi0
N    10.20.7.104/32        [2] area: 4.4.4.4
                           via 172.0.9.104, hsi0
N    10.20.7.199/32        [11] area: 4.4.4.4
                           via 10.20.8.199, trk1
                           via 10.20.9.199, trk2
N    172.0.9.0/24          [1] area: 4.4.4.4
                           directly attached to hsi0
```

5.3.3 Baseline Evaluation: Non HiperSockets Guest Routing Table Entries

The routing entries used in this section were obtained from the LITSHA22 guest, which does not have any HiperSockets interfaces. This guest was specifically chosen to demonstrate that all non-HiperSockets guests, regardless of CPC residency, will always use the directly connected data center interfaces (named trk1 and trk2) to reach another guest's VIPA (see the first four 10.20.7.nnn/32 routing entries in the routing table below), as was previously mentioned in section 5.2 Evaluation Discussion, bullet 1. b), on page 12.

We also see that while this guest does not have a directly connected HiperSockets interface, it is receiving six equal cost [11] indirect routes to the HiperSockets subnet (172.0.9.0/24). Furthermore, we can see that the shortest paths (routes) to this HiperSockets subnet is via this guest's data center interfaces, named trk1 and trk2 on the 10.20.8.0/24 and 10.20.9.0/24 subnets, respectively. Therefore, while this guest resides on a CPC that has a HiperSockets subnet, this guest can still indirectly access the HiperSockets network, if necessary, despite not having a HiperSockets interface.

Also, it should be noted that each guest's VIPA is always be directly attached to the respective guest's dummy0 interface, as expected.

```
        litsha22> sho ip ospf route

        ============ OSPF network routing table ============

<<< Routing entries that are common to all guests were omitted for brevity >>>

        N    10.20.7.69/32        [11] area: 4.4.4.4
                                  via 10.20.8.69, trk1
                                  via 10.20.9.69, trk2
        N    10.20.7.100/32       [11] area: 4.4.4.4
                                  via 10.20.8.100, trk1
                                  via 10.20.9.100, trk2
        N    10.20.7.101/32       [11] area: 4.4.4.4
                                  via 10.20.8.101, trk1
                                  via 10.20.9.101, trk2
        N    10.20.7.104/32       [11] area: 4.4.4.4
                                  via 10.20.8.104, trk1
                                  via 10.20.9.104, trk2
        N    10.20.7.199/32       [1] area: 4.4.4.4
                                  directly attached to dummy0
        N    172.0.9.0/24         [11] area: 4.4.4.4
                                  via 10.20.8.104, trk1
                                  via 10.20.9.104, trk2
                                  via 10.20.8.101, trk1
                                  via 10.20.9.101, trk2
                                  via 10.20.8.100, trk1
                                   via 10.20.9.100, trk2
```

At this point we have compared routing entries for both HiperSockets and non-HiperSockets guests that are all located on the same CPC (SSI cluster member). The next scenario will make comparisons between guests that are located across two different CPCs.

5.4 Live Guest Relocation Evaluations

This section of the paper discusses the results from using Live Guest Relocation to relocate a HiperSockets guest across SSI cluster members to demonstrate that some level, either dynamic or manual, of CPC awareness is required when relocating HiperSockets guests across SSI cluster members.

Once again, we begin with all five guests located on the VM9 SSI Cluster member. In the first relocation scenario we will use the Live Guest Relocation feature to move the LITDCON1 guest over to the VM2 SSI Cluster member. However, for this first scenario we intentionally allow this HiperSockets guest (LITDCON1) to be relocated so that it will have the wrong HiperSockets interface active (up) on the target SSI cluster member, as is illustrated in Figure 3: Live Guest Relocation with wrong HiperSockets interface active on the target SSI cluster, below. In other words, we intentionally disable our REXX and Bash automation scripts to remove all CPC awareness for this particular Live Guest Relocation scenario so that the LITDCON1 guest will have the hsi0 HiperSockets interface active on the VM2 SSI cluster member instead of having the hsi1 Hipersockets interface active, as it should.

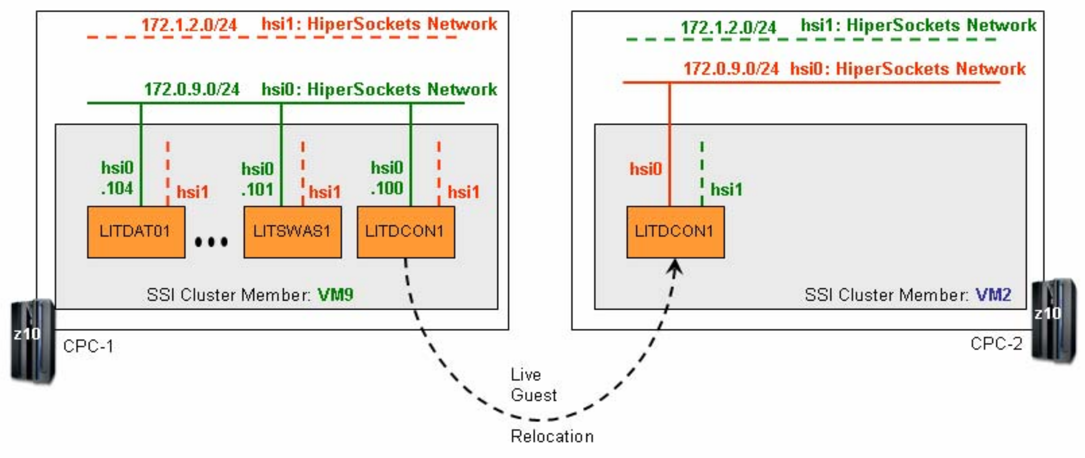


Figure 3: ***Live Guest Relocation with wrong HiperSockets interface active on the target SSI cluster***

In the second HiperSockets guest relocation scenario we continued to use the Live Guest Relocation feature to relocate the LITDCON1 guest from the VM9 SSI Cluster member to the VM2 SSI Cluster member. However, this time our *CPC Awareness* automation scripts are active and therefore, these scripts ensure that the LITDCON1 guest will have the correct HiperSockets interface active (up) once it has been relocated on the destination SSI Cluster member, as is illustrated in Figure 4: *Live Guest Relocation with correct HiperSockets interface active on the target SSI cluster* on page 17.

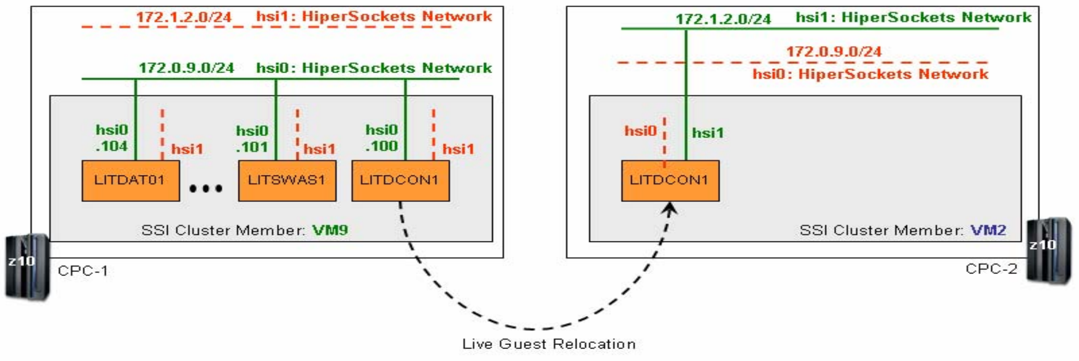


Figure 4: ***Live Guest Relocation with correct HiperSockets interface active on the target SSI cluster***

5.4.1 LITDCON1 – LGR with the wrong HiperSockets Interface active on the target member

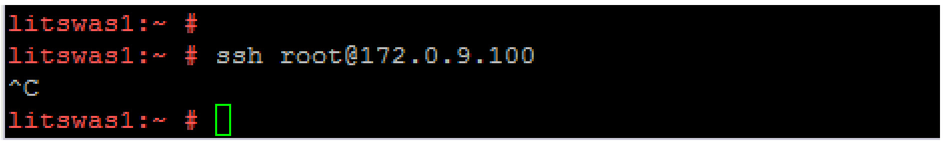
This scenario involves relocating a guest (LITDCON1) with HiperSockets interfaces across SSI Cluster members. For this scenario, we intentionally allow this guest to continue using the same HiperSockets interface (hsi0) on the target SSI Cluster member (VM2) that it was using on the source SSI cluster member (VM9) to illustrate the routing issues that result when a HiperSockets guest is not CPC aware, and therefore has the wrong HiperSockets interface active when it has been relocated to the target member.

To illustrate the routing corruption that results from having the wrong HiperSockets interface up, we will first show the routing table entries from the LITSWAS1 HiperSockets guest that is still located on the other SSI Cluster member, VM9. This guest's routing table still shows that it is directly attached to the 172.0.9.0/24 HiperSockets subnet, as it should be.

```
N    172.0.9.0/24         [1] area: 4.4.4.4
                          directly attached to hsi0
```

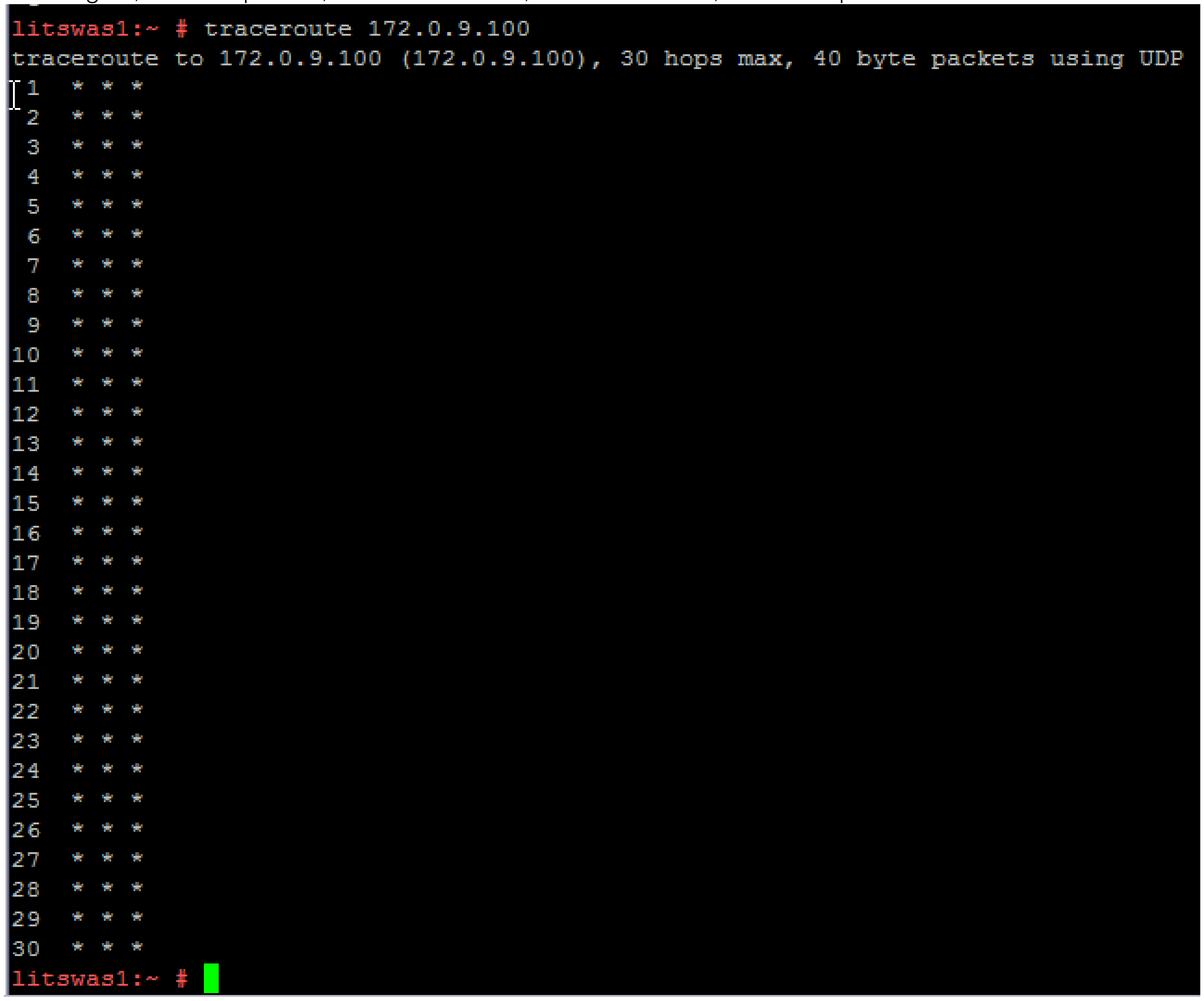
However, when we try to establish a connection from the LITSWAS1 guest to the LITDCON1 guest across the HiperSockets network the connection attempt hangs, as expected, because a HiperSockets network cannot span CPCs as was illustrated in Figure 1: *Duplicate HiperSockets subnets across multiple CPCs* on page 5.

litswas1:~ # ssh root@172.0.9.100



```
litswas1:~ #
litswas1:~ # ssh root@172.0.9.100
^C
litswas1:~ #
```

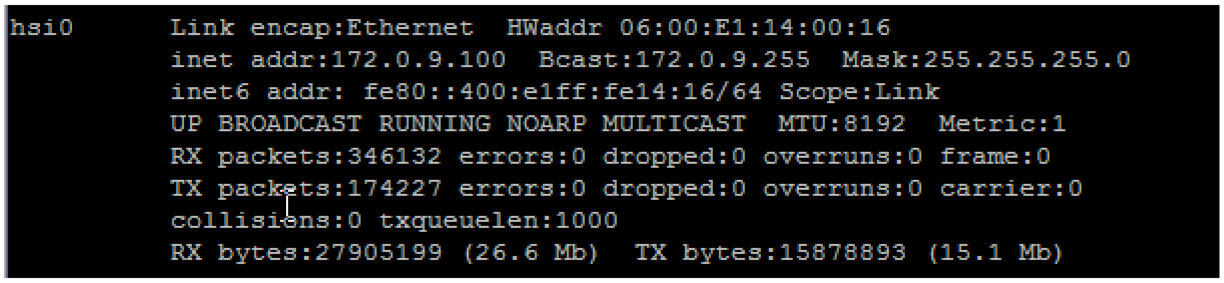
Next, when we try to trace the route we see that there really is no path to the destination, despite the routing table indicating that the guest has a directly attached interface on the respective HiperSockets subnet. Once again, this is expected; it is not desirable, but nonetheless, it is still expected.

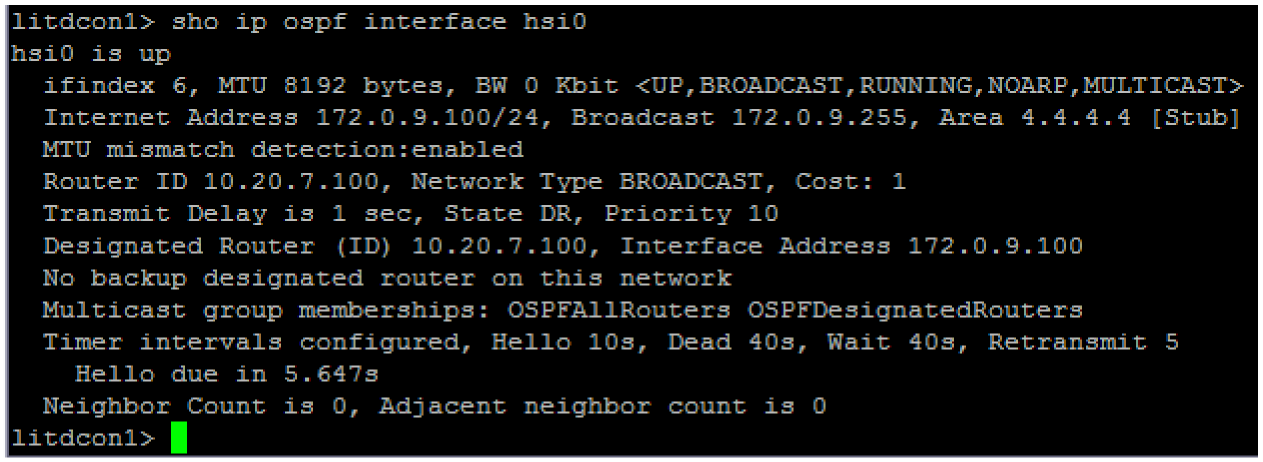


```
litswas1:~ # traceroute 172.0.9.100
traceroute to 172.0.9.100 (172.0.9.100), 30 hops max, 40 byte packets using UDP
 1  * * *
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  * * *
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
30  * * *
litswas1:~ #
```

When we examine the routing table on the LITDCON1 guest that was relocated to the other SSI Cluster member (VM2), we would expect to see an entry indicating that the guest has a directly attached interface on the 172.0.9.0/24 HiperSockets subnet. Under normal conditions a directly attached HiperSockets interface would be listed, but because this is not a normal condition, no such directly attached hsi0 interface entry exists and therefore, we cannot show you what was not listed when we displayed the OSPF routing table.

The next two displays indicate that the guest's hsi0 interface is up and that the OSPF process claims to be using it, but regardless, LITDCON1 is not actually able to use it.


```
hsi0      Link encap:Ethernet  HWaddr 06:00:E1:14:00:16
          inet addr:172.0.9.100  Bcast:172.0.9.255  Mask:255.255.255.0
          inet6 addr: fe80::400:e1ff:fe14:16/64 Scope:Link
          UP BROADCAST RUNNING NOARP MULTICAST  MTU:8192  Metric:1
          RX packets:346132 errors:0 dropped:0 overruns:0 frame:0
          TX packets:174227 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:27905199 (26.6 Mb)  TX bytes:15878893 (15.1 Mb)
```


```
litdcon1> sho ip ospf interface hsi0
hsi0 is up
  ifindex 6, MTU 8192 bytes, BW 0 Kbit <UP,BROADCAST,RUNNING,NOARP,MULTICAST>
  Internet Address 172.0.9.100/24, Broadcast 172.0.9.255, Area 4.4.4.4 [Stub]
  MTU mismatch detection:enabled
  Router ID 10.20.7.100, Network Type BROADCAST, Cost: 1
  Transmit Delay is 1 sec, State DR, Priority 10
  Designated Router (ID) 10.20.7.100, Interface Address 172.0.9.100
  No backup designated router on this network
  Multicast group memberships: OSPFAllRouters OSPFDesignatedRouters
  Timer intervals configured, Hello 10s, Dead 40s, Wait 40s, Retransmit 5
    Hello due in 5.647s
  Neighbor Count is 0, Adjacent neighbor count is 0
litdcon1>
```

Another misleading diagnostic is seen when we examine the Linux kernel's IP routing table, which has an entry for the HiperSockets interface. Clearly, this is not a reliable diagnostic metric, especially when the guest is relying on the OSPF process to satisfy its routing requirements. This is another reason why we prefer to use the OSPF routing table rather than the Linux kernel's routing table.

```
litdcon1:~ # route -n
         Kernel IP routing table
         Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
         172.0.9.0       0.0.0.0         255.255.255.0   U     0      0        0 hsi0
```

5.4.2 LITDCON1 – LGR with the correct HiperSockets Interface active on the target member

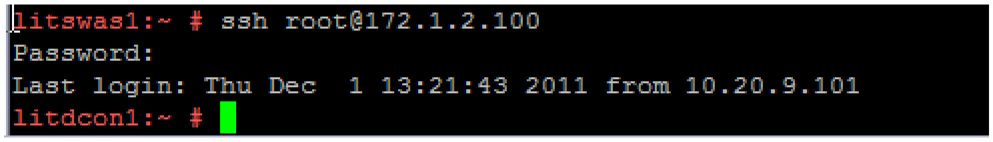
So, what happens when our automation ensures that a relocated guest has the correct HiperSockets interface up? This section presents the routing table entries from various guests within the OSPF area, as well as a screen capture for a connection attempt when the correct HiperSockets interface is active on the target member for a guest that was recently relocated.

Once again, the LITDCON1 guest was relocated from the VM9 to the VM2 SSI cluster member, but this time our automation scripts ensured that LITDCON1's hsi1 HiperSockets interface was active (up) and the hsi0 HiperSockets interface was configured down when it was relocated to the VM2 Cluster member.

For this scenario we begin by examining the routing table entries on the LITSWAS1 guest located on the VM9 SSI Cluster member. This time we see that in addition to the original directly attached HiperSockets interface (hsi0), this guest is now receiving two equal cost indirect routing entries to reach the HiperSockets network which we know is located on the other CPC.

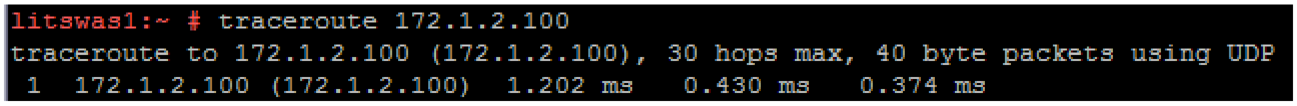
```
N    172.0.9.0/24         [1] area: 4.4.4.4
                          directly attached to hsi0
N    172.1.2.0/24         [11] area: 4.4.4.4
                          via 10.20.9.100, trk2
                          via 10.20.8.100, trk1
```

This time, when we try to establish an SSH session with LITDCON1's HiperSockets interface, it succeeds, as expected. Note that since the correct HiperSockets interface is now up on the LITDCON1 guest, we specifically use the guest's host address on the correct HiperSockets network to establish the SSH connection demonstrating that the HiperSockets interface is now functional. It is also worth noting that the VIPA for LITDCON1 remains active, as it should, during the move, even though the IP address on the underlying HiperSockets network changes.



```
litswas1:~ # ssh root@172.1.2.100
Password:
Last login: Thu Dec  1 13:21:43 2011 from 10.20.9.101
litdcon1:~ #
```

Also, and just for completeness, tracing the route from LITSWAS1 to LITDCON1's HiperSockets interface is now successful.



```
litswas1:~ # traceroute 172.1.2.100
traceroute to 172.1.2.100 (172.1.2.100), 30 hops max, 40 byte packets using UDP
 1  172.1.2.100 (172.1.2.100)  1.202 ms   0.430 ms   0.374 ms
```

6 Conclusion

We have shown that Live Guest Relocation is transparent for guests that do not have HiperSockets interfaces while the same is not true for guests with HiperSockets interfaces.

Because HiperSockets do not span CPCs, we have also shown that special attention is needed when utilizing the Live Guest Relocation feature in z/VM 6.2 to actively relocate guests with HiperSockets to ensure that routing loops will not be injected into an OSPF area.

Our approach to utilize REXX EXECs and a Bash script to make a HiperSockets guest CPC aware is an example of the special attention needed to preserve a data center’s high availability that relies on OSPF and VIPA, when actively relocating a HiperSockets guest across SSI cluster members using z/VM’s Live Guest Relocation feature.

Appendix – Reference Configuration files

6.1 REXX EXECs

NOTE: The scripts and sample OSPFD and ZEBRA configuration files contained in this paper's appendix are neither supported by IBM, nor supplied with any IBM product. These scripts and sample configuration files were created from scratch, with the sole intention to support the testing described in this paper.

6.1.1 LGREVACA

see footnote 1 below

```
/* REXX */
/*                                                                   */
/*  This EXEC requires that a guest workload parameter be passed in  */
/*    when it is called that will be used to match guest ids that are */
/*    running on the current z/VM host (where this EXEC is running)  */
/*                                                                   */
/*  This EXEC will then use the workload parameter to determine which */
/*    destination z/VM host the guest workloads will need to be moved */
/*    to.                                                            */
/*                                                                   */
/*  This EXEC will then call another REXX EXEC that will perform     */
/*    the actual Live Guest Relocations, by passing that EXEC both the */
/*    guest workload and destination z/VM host values.               */
/*                                                                   */
/*  Valid guest workload parameter values that can be passed in are:  */
/*    ECM  -to evacuate just the ECMnnnnn workload guests on this host */
/*    LIT  -to evacuate just the LITnnnnn workload guests on this host */
/*    TIV  -to evacuate just the TIVnnnnn workload guests on this host */
/*    ALL  -to evacuate ALL the workload guests running on this host  */
/*                                                                   */

ADDRESS COMMAND

PARSE UPPER ARG WORKLOAD ADMIN_IN JUNK

if WORKLOAD = ''               /*  test for a blank workload       */
  then Call Bad_workload_passed
else WORKLOAD=SUBSTR(WORKLOAD,1,3) /* trim to the first 3 chars    */


if ADMIN_IN = ''               /*  test for a blank admin id,      */
  then ADMIN_IN='RJBRENN'      /*  set it to a known value if blank */
else ADMIN_IN=SUBSTR(ADMIN_IN,1,8) /* only accept the first 8 chars  */

/******************/
/******************/
/**              **/
/**  Main logic  **/
/**               **/
/*****************/
/*****************/

CALL Init_vars                /* initilaize the exec's variables       */
CALL Where_am_i               /* determine where this exec is running  */
CALL Set_logfile
CALL Write_header

Select

  when WORKLOAD = 'ECM' then
    do
      call Set_ECM_Dest
      call LGREVACB WORKLOAD DEST Log_file
      call Send_logfile
    end
  when WORKLOAD = 'LIT' then
    do
      call Set_LIT_Dest
      call LGREVACB WORKLOAD DEST Log_file
      call Send_logfile
    end
  when WORKLOAD = 'TIV' then
    do
      call Set_TIV_Dest
      call LGREVACB WORKLOAD DEST Log_file
      call Send_logfile
    end
  when WORKLOAD = 'ALL' then
    do
      call Set_LIT_Dest
      WORKLOAD = 'LIT'
      call Set_logfile
      call LGREVACB WORKLOAD DEST Log_file
      call Send_logfile

      call Set_ECM_Dest
      WORKLOAD = 'ECM'
      call Set_logfile
      call LGREVACB WORKLOAD DEST Log_file
      call Send_logfile

      call Set_TIV_Dest
      WORKLOAD = 'TIV'
      call Set_logfile
      call LGREVACB WORKLOAD DEST Log_file
      call Send_logfile
    end

  otherwise call Bad_workload_passed
```

```
end   /* select end */

'CP DET 999'                   /* detach the log file temp disk       */

EXIT 0                        /* end of main program                 */

/*---------------------------------------------------------------*/

/**********************/
/**********************/
/**                 **/
/**    SUBROUTINE    **/
/**      SECTION     **/
/**                 **/
/**********************/
/**********************/

/*---------------------------------------------------------------*/

Init_vars:

 cur_host='LIMBO'              /* initialize some control variables    */

                               /* Set the valid host array index.      */
 ECM_host_index = 2            /*   As more hosts are added to the     */
 ECM_host.1 = 'LTICVM2'        /*   data center, increase the index    */
 ECM_host.2 = 'LTICVM9'        /*   and add another stem variable      */
 /* ECM_host.3 = 'LTICVMn' */

                               /* Set the valid host array index.      */
 LIT_host_index = 2            /*   As more hosts are added to the     */
 LIT_host.1 = 'LTICVM2'        /*   data center, increase the index    */
 LIT_host.2 = 'LTICVM9'        /*   and add another stem variable      */
 /* LIT_host.3 = 'LTICVMn' */
                               /* Set the valid host array index.      */
 TIV_host_index = 2            /*   As more hosts are added to the     */
 TIV_host.1 = 'LTICVM4'        /*   data center, increase the index    */
 TIV_host.2 = 'LTICVM5'        /*   and add another stem variable      */
 /* TIV_host.3 = 'LTICVMn' */

                               /* create a temp disk to                */
 'CP DEF VFB-512 999 BLK 1024' /* hold the logfile                     */
  IF RC = 0 THEN
   DO
     QUEUE '1'
     QUEUE 'TMP999'
     'FORMAT 999 X'
   END
  ELSE
   DO
     SAY "ABORT: Unable to create temp disk 999 filemode X"
     EXIT 856
   END

return

/*-----------------------------------------------------*/

Where_am_i:

  'EXECIO * CP (STEM VM_HOST. STRING Q USERID'
  cur_host=WORD(VM_HOST.1,3)        /* extract the 3rd word from the  */
                                    /*    first line returned from    */
                                    /*    the Q USERID command        */
return

/*-----------------------------------------------------*/

Set_logfile:

  Day = WORD(DATE(),1)
  Month = TRANSLATE(WORD(DATE(),2))
  Year = WORD(DATE(),3)

  Cur_time = TIME()
  Hrs = SUBSTR(TIME(),1,2)
  Min = SUBSTR(TIME(),4,2)
  Sec = SUBSTR(TIME(),7,2)

    /*  Workload = 3 chars (ECM, TIV, or LIT)                         */
    /*  Concat the 3 char WORKLOAD with the 2 char month + 2 char day  */
    /*  to end up with a 7 character log file name that will map to    */
    /*  a log file specific to the workload for the respective day.    */
    /*  File uniqueness is established by the Log_file_type var below  */
  Log_file_name = WORKLOAD||Month||Day

    /*  LGR Log file type will begin with the characters 'LG'          */
    /*  concatenated with the Hour, Minute, and Second that the        */
    /*  log file is created.  This will ensure log file uniquess       */
    /*  in the event that a worklooad is moved more than once per day. */
  Log_file_type = 'LG'Hrs||Min||Sec

  Log_file = Log_file_name||' '||Log_file_type||' X1'

return

/*-----------------------------------------------------*/

Write_header:

  L1 = '  Log file name: 'Log_file
  L2 = '  Administration ID: 'ADMIN_IN
  L3 = ' '
  L4 = '  Time  | '
  L5 = '  Stamp | Relocation Message'
  L6 = '--------+-------------------------------------------------'

  call LINEOUT Log_file,L1
  call LINEOUT Log_file,L2
  call LINEOUT Log_file,L3
  call LINEOUT Log_file,L4
  call LINEOUT Log_file,L5
  call LINEOUT Log_file,L6

return

/*----------------------------------------------------*/

Set_ECM_Dest:

  DO k=1 to ECM_host_index
   if ECM_host.k <> cur_host then DEST = ECM_host.k
   else nop;
  END /* end do */

return

/*----------------------------------------------------*/

Set_LIT_Dest:
  DO k=1 to LIT_host_index
   if LIT_host.k <> cur_host then DEST = LIT_host.k
   else nop;
  END /* end do */

return

/*----------------------------------------------------*/

Set_TIV_Dest:

  DO k=1 to TIV_host_index
   if TIV_host.k <> cur_host then DEST = TIV_host.k
   else nop;
  END /* end do */

return

/*--------------------------------------------------------------*/

Send_logfile:

  CALL LINEOUT Log_file       /* close the log file   */

        /*  unconditionally send the system programmers a      */
        /*   copy of the log file regardless of who runs       */
```

```
      /*   this Live Guest Relocation Exec                   */

  'EXEC SENDFILE 'Log_file' THEGREEK'
  'EXEC SENDFILE 'Log_file' RJBRENN'

      /*  only send the current user a copy of the log if the */
      /*  admin_in that was passed in DOES NOT match one of   */
      /*  the reserved system programmer ids                  */

  select
    when (ADMIN_IN <> 'RJBRENN') | ,
         (ADMIN_IN <> 'THEGREEK')

      then 'EXEC SENDFILE 'Log_file' 'ADMIN_IN

    otherwise; /* null...already sent log file to the sysprogs      */
  end

return

/*--------------------------------------------------------------*/

Bad_workload_passed:

  L1= ' A bad workload parameter was passed in'
  L2= ' Workload value passed in was ->'WORKLOAD'<--'
  L3= '  by operations manager - exiting with a RC 86'
  L4= ''
  CALL LINEOUT Log_file,L1
  CALL LINEOUT Log_file,L2
  CALL LINEOUT Log_file,L3
  CALL LINEOUT Log_file,L4

  CALL Send_logfile           /* send the log file    */

  'CP DET 999'

  exit 86                /* abort                              */

return                   /* end of  subroutine                 */
```

6.1.2 LGREVACB

see footnote 1 below

```
/* REXX */
ADDRESS COMMAND

PARSE ARG Workload_in Dest_in Log_file_in

/******************/
/******************/
```

```
/**              **/
/**  Main logic  **/
/**              **/
/******************/
/******************/

CALL Init_vars               /* initilaize the exec's variables       */
CALL Where_am_i              /* set the current host variable         */
CALL Build_list              /* build the list of guests              */

                             /* now that we know how many users we're */
stem.failed_result=THEUSERS.I /* dealing with, we can set the index    */
stem.moved_result=THEUSERS.I  /* for the reporting variable arrays     */

/**********************/
/*                    */
/*  Start processing  */
/*  the guest list    */
/*                    */
/**********************/

DO I=1 TO THEUSERS.0         /* start processing the guest list       */

  cur_guest = THEUSERS.I     /* set current working guest variable    */

  hsi_flag = 0               /* reset/initialize all the working      */
  Failed_flag  = 0           /* variables for the current guest       */
  Success_flag = 0           /* that is being moved in this iteration */

  CALL TEST_4_HSI            /* determine if this current guest       */
                             /*  has a HiperSocketsinterface, or not  */
  if hsi_flag = 1 then       /* This guest has a hipersocket, so we   */
    CALL Start_linux_script  /*  need to shut it down before the move */
  else nop;                  /* else this is a non hsi guest          */

  CALL Detach_devs

  CALL Move_guest            /* try to move the current guest         */

  CALL Verify_evac_msgs      /* interrogate CP messages to determine  */
                             /* if the current guest moved or not     */

  Select                     /* determine results from the relocation */
    when Success_flag = 1    /* a successful move message was receive */
      then CALL EVACUATE_SUCCEEDED  /* update the success variables    */
    otherwise CALL EVACUATE_FAILED  /* move failed, update failed vars */
  end  /* end select  */

                             /* regardless if the guest moved, or not */
  if hsi_flag = 1 then       /*  if guest has hsi, then we need to    */
    CALL Set_LGR_control_file /*  signal the guest to ifup the right   */
                             /*  hsi based on which host the guest    */
```

```
                              /*  currently resides on                */
  else nop;                   /* else continue, his is a regular guest */

 END                          /* end do I to THEUSERS.0 loop          */

 CALL Report_Results          /* echo results to the screen           */

EXIT 0                        /* end of main program                  */

/*------------------------------------------------------------*/

/**********************/
/**********************/
/**                 **/
/**    SUBROUTINE    **/
/**      SECTION     **/
/**                 **/
/**********************/
/**********************/

/*------------------------------------------------------------*/

Init_vars:

 new_host=Dest_in              /* control variable used by subroutine  */

 cur_host='LIMBO'              /* control variable used by subroutine  */
 CP_send_host = 'LIMBO'        /* if evac was successful, then the */
 cur_guest='LIMBO'             /* control variable used by subroutine  */
 Cur_time = TIME()             /* initialize with current time         */

 hsi_flag = 0                  /* HiperSocketsflag variable            */
 failed_count = 0              /* reporting variable                   */
 success_count =0              /* reporting variable                   */
 valid_user_count=0            /* variable to hold the # of valid users*/
 total_evacs = 0               /* counts # of guests evacuated         */
 Failed_flag  = 0
 Success_flag = 0
 Evac_msg  = ''

return

/*-----------------------------------------------*/

Where_am_i:

  'EXECIO * CP (STEM VM_HOST. STRING Q USERID'
  cur_host=WORD(VM_HOST.1,3)        /* extract the 3rd word from the  */
                                    /*    first line returned from    */
                                    /*    the q userid command        */
  CP_send_host = cur_host           /* set target for cp send command */
  Cur_time = TIME()                /* Obtain current time            */
  Msg_1 = Cur_time' Workload: 'Workload_in

  Cur_time = TIME()                /* Obtain current time            */
  Msg_2 = Cur_time' Current Host: 'cur_host

  CALL LINEOUT Log_file_in,Msg_1
  CALL LINEOUT Log_file_in,Msg_2

return

/*----------------------------------------------------*/

Build_list:

  Cur_time = TIME()                /* Obtain current time            */
  Msg_1 = Cur_time' Building the Workload guest list that...'

  Cur_time = TIME()                /* Obtain current time            */
  Msg_2 = Cur_time' + matches the pattern of    --> 'Workload_in

  Cur_time = TIME()                /* Obtain current time            */
  Msg_3 = Cur_time' + and currently reside on   --> 'cur_host

  Cur_time = TIME()                /* Obtain current time            */
  Msg_4 = Cur_time' + that will be relocated to --> 'new_host

  CALL LINEOUT Log_file_in,Msg_1
  CALL LINEOUT Log_file_in,Msg_2
  CALL LINEOUT Log_file_in,Msg_3
  CALL LINEOUT Log_file_in,Msg_4
   /*                                                */
  'PIPE CP Q N AT *',
       '| SPLIT , ',
       '| STRIP ',
       '| CHOP 8 ',
       '| LOCATE /'Workload_in'/ ',
       '| STEM THEUSERS. '

                                  /* if no users matching the pattern  */
  found_user = 0                  /* were found using the PIPE command */
                                  /* then the stem variable called     */
  DO k=1 to THEUSERS.0            /* 'theusers.0' will be empty/null   */
     found_user = found_user + 1  /* i.e.:theusers.1 will not exist    */
  END                             /* thus found_user count will never  */
                                  /* increment so the do i=1  loop     */
                                  /* would not be entered.             */
if found_user =  0
 then
   do
    Cur_time = TIME()                /* Obtain current time            */
    Msg_5 = Workload_in' is not running on this host ('cur_host')'
    Msg_5 = Cur_time''Msg_5

    CALL LINEOUT Log_file_in,Msg_5

    Msg_6 ='Exiting the EXEC from within the Build List routine. RC:14'
    Msg_6 = Cur_time''Msg_6

    CALL LINEOUT Log_file_in,Msg_6
    CALL LINEOUT                  /* close the log file                */
    exit 14
   end
  else;                            /* guests matching the filter       */
                                   /* were found on this host          */
                                   /* return to main program flow      */
return

/*---------------------------------------------------*/

Move_guest:

            /* This routine attempts to move the    */
            /*   current guest.                     */
            /*                                      */
            /* NOTE: the STEM LINES. variable will  */
            /* contain the CP messages received     */
            /* when this routine issues the         */
            /* VMRELO MOVE command.  These          */
            /* messages will be interrogated by the */
            /* Verify_evac_msgs routine             */

  Msg_1=' Attempting to move 'cur_guest' from 'cur_host' to 'new_host
  Msg_1=Cur_time''Msg_1

  CALL LINEOUT Log_file_in,Msg_1

  'EXECIO * CP (STEM LINES. STRING VMRELO MOVE 'cur_guest' 'new_host

  total_evacs = total_evacs + 1

return

/*---------------------------------------------------*/

Detach_devs:

  Cur_time = TIME()                /* Obtain current time            */
  Msg_1 = ' Detaching devices 190-19F on '
  Msg_1 = Cur_time''Msg_1' 'cur_guest

  CALL LINEOUT Log_file_in,Msg_1

  'CP FOR 'cur_guest' CMD DET 190-19F'
```

```
return

/*---------------------------------------------------*/

TEST_4_HSI:                  /* subroutine to determine if the guest */
                             /* being moved is one with HiperSockets */
 Cur_time = TIME()           /* Obtain current time                  */
 Msg_1='- - -'
 CALL LINEOUT Log_file_in,Msg_1

 select
   when (cur_guest = 'LITDCON1' | ,
         cur_guest = 'LITSWAS1' | ,
         cur_guest = 'LITDAT01')
   then hsi_flag = 1
   otherwise hsi_flag = 0
 end

  hsi_msg=cur_guest' has a HiperSockets Interface'

  if hsi_flag = 1 then        /*  if guest has hsi, then we need to    */
     hsi_msg=cur_guest' Has a HiperSockets Interface'
  else
     hsi_msg=cur_guest' Does NOT have a HiperSockets Interface'

  Cur_time = TIME()                /* Obtain current time            */
  hsi_msg = cur_time' 'hsi_msg

  CALL LINEOUT Log_file_in,hsi_msg

return

/*---------------------------------------------------*/

Start_linux_script:
  /* It was detemrined that this guest has a HiperSocketsinterface    */
  /*   therefore we need to start the BASH script on the guest        */
  /*   while it still resides on this z/VM host                       */


  lnxcmd1 = 'cd /var/log/lxconusr'  /* lower case linux commands to   */
  lnxcmd2 = './lgr_hsi.sh &'        /* be issued on the guest console */

  'CP SEND 'cur_guest' AT 'CP_send_host' 'lnxcmd1

  'CP SLEEP 3 SEC'                  /* give the guest time to process */
                                    /* the last command passed to it  */

  'CP SEND 'cur_guest' AT 'CP_send_host' 'lnxcmd2

  Cur_time = TIME()                /* Obtain current time            */
  Msg_1 = ' The Bash script is now running on 'cur_guest
  Msg_1 = Cur_time''Msg_1

  CALL LINEOUT Log_file_in,Msg_1

  'CP SLEEP 3 SEC'                  /* give the guest time to       */
                                    /* initialize the script        */

     /*--------------------------------------------*/
     /*  at this point the script running on the guest */
     /*  will unconditionally 'ifdown' all hsi         */
     /*  interfaces, it will also echo a '0' into the  */
     /*  relocation control file, and the script will  */
     /*  spin until the relocation control file        */
     /*  contains a '1'                                */
     /*                                                */
     /*  There are two ways that the relocation control*/
     /*  file can contain a '1'                        */
     /*                                                */
     /*  1) if the guest is successfully relocated,    */
     /*     the script running on the guest will detect*/
     /*     that the guest has been moved and will     */
     /*     echo a '1' into the relocation control     */
     /*     file to signal the guest to 'ifup' the     */
     /*     appropriate hsi interfaces                 */
     /*                                                */
     /*  2) this exec determined that the relocate     */
     /*     failed and as such, this exec will echo    */
     /*     a '1' into the control file located on the */
     /*     guest in order to signal the guest that it */
     /*     must 'ifup' the appropriate hsi interfaces */
     /*                                                */
     /*  Regardless if the guest was moved or not,     */
     /*  the script running on the guest will 'ifup'   */
     /*  the proper hsi interfaces based on which      */
     /*  host the guest resides on at the time the     */
     /*  relocation control file contains a '1'.       */
     /*                                                */
     /*  ------------------------------------------   */

return

/*--------------------------------------------------*/

Verify_evac_msgs:             /* this routine has a dependency on a   */
                              /* variable loaded by the Move_guest    */
                              /* routine.  Specifically, STEM LINES.  */
                              /* contains the CP messages received    */
                              /* when the Move_guest routine issues   */
                              /* the VMRELO MOVE command              */
  Success_flag = 0
  Failed_flag = 0
  Do n = 1 To LINES.0
    Evac_msg = LINES.n
                              /* interrogate the CP messages from */
                              /* the relocate move command        */
    EVAC_SUCCESS = WORDPOS('has been relocated from',LINES.n)

    if EVAC_SUCCESS <> 0   /* then the success msg string was found   */
      then Success_flag = 1
      else Failed_flag = 1               /* move failed              */

  end /* end do n=1 to LINES.0  */

return

/*------------------------------------------------------------*/

EVACUATE_SUCCEEDED:

  Cur_time = TIME()               /* Obtain current time           */

  failed_result.I=' n/a   '
  moved_result.I='Success'

  CP_send_host = new_host         /* if evac was successful, then the */
                                  /*  current host changed so we need */
                                  /* to change the destination of     */
                                  /* subsequent CP send commands      */
                                  /* re: set_lgr_control_file routine */

  success_count = success_count + 1

  Do p = 1 To LINES.0
    CALL LINEOUT Log_file_in,Cur_time' 'LINES.p
  end

  'CP SLEEP 10 SEC'          /* give the guest console time to        */
                             /*  initialize on the new host           */
return

/*------------------------------------------------------------*/

EVACUATE_FAILED:

  Cur_time = TIME()               /* Obtain current time           */

  CP_send_host = cur_host         /* just ensuring that the dest of   */
                                  /*  subsequent CP send commands     */
                                  /*  target the correct host         */

  failed_result.I='FAILED '
  moved_result.I='  n/a  '
  failed_count = failed_count + 1


  Do p = 1 To LINES.0
    CALL LINEOUT Log_file_in,Cur_time' 'LINES.p
  end

  /*  guest console will remain connected since guest did not move  */
  /*  therefore, no need to pause the script so it can initialize   */
  /*  as is needed when the guest actually moves to a new host      */

return    /* end of evacuate failed subroutine                  */

/*-----------------------------------------------------------------*/

Set_LGR_control_file:

      /* NEED TO INFORM THE SHELL SCRIPT RUNNING ON THE       */
      /* GUEST THAT IT NEEDS TO STOP LOOPING AND TO           */
      /* ifup THE HIPERSOCKETSINTERFACE                       */
      /*  - This is especially important for the case where   */
      /*    the guest does not move.  If the guest moves,     */
      /*    the bash script running on the guest will detect  */
      /*    the move and will break out of the loop by itself */

  Cur_time = TIME()                /* Obtain current time            */
  Msg_1 = ' Echo >1 into the LGR control File was issued on 'cur_guest
  Msg_1 = Cur_time''Msg_1

  CALL LINEOUT Log_file_in,Msg_1

           /* set up the lowercase linux command to be  */
           /* issued on the guest console               */
  lnx_echo_cmd = 'echo 1 > /var/log/lxconusr/lgr_control_file'

           /*  send the hsi reset signal to the guest on the right host*/
  'CP SEND 'cur_guest' AT 'CP_send_host' 'lnx_echo_cmd

return                 /* end of  subroutine                  */

/*-----------------------------------------------------------------*/

Report_Results:

 CALL LINEOUT Log_file_in,' '

 stem.RL=14             /* Report header Array index            */

 RL.1= '--------------------------------------------------+--------'
 RL.2= '                 Relocation Report                | count  '
 RL.3= '--------------------------------------------------+--------'
 RL.4= 'Number of Guests that SUCCESSFULLY moved was     | 'success_count
 RL.5= '-----------------------------------------------|--------'
 RL.6= 'Number of Guests that were NOT moved was         | 'failed_count
 RL.7= '-----------------------------------------------+--------'
 RL.8= 'Total Relocation attempts (success + failed) was | 'total_evacs
 RL.9= ''

 RL.10=' '
 RL.11='     Itemized list of results: '
 RL.12='-----------------------------------'
 RL.13='Guest id |    Evacuation result     |'
 RL.14='---------+-------------+-------------'

 /* DO q=1 TO RL.0            start procesing the report summary  */
 DO q=1 TO 14                  /* start procesing the report summary  */
   CALL LINEOUT Log_file_in,RL.q

   /*     THEUSERS.m' |  'moved_result.m'    |  'failed_result.m'   |' */
   /* SAY THEUSERS.m' |  'moved_result.m'    |  'failed_result.m'   |' */

 END /* end do */

 DO m=1 TO THEUSERS.0                /* start procesing the guests    */
   lgr_rslt=THEUSERS.m' |  'moved_result.m'    |  'failed_result.m'   |'
   CALL LINEOUT Log_file_in,lgr_rslt

   /*     THEUSERS.m' |  'moved_result.m'    |  'failed_result.m'   |' */
   /* SAY THEUSERS.m' |  'moved_result.m'    |  'failed_result.m'   |' */

 END /* end do */

   CALL LINEOUT Log_file_in,' '
   CALL LINEOUT Log_file_in,'  End of Report  '
   CALL LINEOUT Log_file_in,' '
   CALL LINEOUT Log_file_in      /* close file  */

return                   /* end of  subroutine                   */

/*-------------------------------------------------------------*/
```

6.2 Bash Script

see footnote 1 below

```
#! /bin/bash
#----------------------
# this script is used for guests that use HiperSockets
#    and that will be moved by LGR.
#    This script determines
#      which VM host the guest resides on and configures
#      the appropriate HiperSockets interface online
#      so that the guest's VIPA (dummy0) interface
#      which maps to the guest's DNS entry,
#      is properly advertised by OSPF.
#
#  Initialize the script's controls
#
VM2=LTICVM2
VM9=LTICVM9
HSI_RC=-86
#
#-----------
#create the control file so that it can't be compromised by a casual user
touch  /var/log/lxconusr/lgr_control_file
#
# initialize the control file
echo 0 >  /var/log/lxconusr/lgr_control_file
#--------------------------------------------------
#
# load vmcp module and give it a few seconds to initialize
#
echo "initializing vmcp engine..."
sudo /sbin/modprobe vmcp
echo 'sleeping 2 seconds---letting modprobe vmcp load'
sleep 2
#
#---------
#
# Get the VM Host and the Linux guest's userid
Q_userid=$(sudo /sbin/vmcp q userid)
Guest_id=${Q_userid:0:8}
Initial_VM_Host=${Q_userid:12:8}
#
#********************************************
#  echo messages to monitor script progress
#
echo "Initial userid   = "$Q_userid
echo "Linux Guest id   = "$Guest_id
echo "Initial VM Host  = "$Initial_VM_Host
echo
#*******************************************
#
#
#   This script was explicitly started,
#    either manually or by Operations Manager,
#    therefore the guest is moving so its time
#    to determine the Guest's original/current VM Host
#
Current_VM_Host=$Initial_VM_Host
#
#  echo a progress report message...
echo "Current VM Host is : "$Current_VM_Host
echo
#
#------
#
# Only deal with the Guests of interest in case this script is
# accidentally run on a guest that does not have hsi interfaces
#
if [ "$Guest_id" = "LITDCON1" ] || [ "$Guest_id" = "LITDAT01" ] || [ "$Guest_id" = "LITSWAS1" ]
   then
```

```
echo "Linux HiperSocketsguest id matched..."$Guest_id
   #    unconditionally config both HiperSockets interfaces
   #    offline (ifdown) to force an OSPF link state update that
   #    makes this guest's VIPA (dummy0)
   #    only accesible via the trk1 and trk2
   #    interfaces for now.
   sudo /sbin/ifdown hsi0
   echo "Ifdown >>> hsi0 <<< issued"
   sudo /sbin/ifdown hsi1
echo "Ifdown >>> hsi1 <<< issued"
Current_VM_Host=$(sudo /sbin/vmcp q userid)
Current_VM_Host=${Current_VM_Host:12:8}
#**********************************************
echo "Current VM Host ="$Current_VM_Host
echo "Initial VM Host ="$Initial_VM_Host
#
# set the loop control variable = to the control file contents
control_file_var=$(</var/log/lxconusr/lgr_control_file)
echo 'script debug echo- control file variable entering loop is = '$control_file_var
echo '--------------------------------------------------------------------------------'
#
# spin until the control file contains a 1
#
while [ $control_file_var = 0 ]
        # while [ $Initial_VM_Host = $Current_VM_Host ]
  do
   #**************************************
    echo "Entered loop..............."
   #**************************************

   control_file_var=`</var/log/lxconusr/lgr_control_file`
   echo 'control var before if-then-else is '$control_file_var

   Current_VM_Host=$(sudo /sbin/vmcp q userid)
   Current_VM_Host=${Current_VM_Host:12:8}

    if [ "$Initial_VM_Host" = "$Current_VM_Host" ]
      then   # the guest has not moved to a new VM host  #
             # set the not moved flag in the control file  #
        echo 0 > /var/log/lxconusr/lgr_control_file
      else # guest moved, set the moved flag in the control file
        echo 1 > /var/log/lxconusr/lgr_control_file
              #Current_VM_Host=$(vmcp q userid)
              #Current_VM_Host=${Current_VM_Host:12:8}
    fi

    sleep 10
    control_file_var=`</var/log/lxconusr/lgr_control_file`
    echo 'control file variable at bottom of loop is --> '$control_file_var

  done
#
#  At this point of the code, the control file contains a '1'
#  It is now time to figure out which hsi interface needs to be up
#------------
#  the guest can reside on one of two z/VM hosts
#  - VM9 resides on H91 and will use the E1 Hipersocket
#    which maps to the E100 devices and the hsi0 interface
```

```
      #
      #  - VM2 resides on R91 and will use the E3 Hipersocket
      #    which maps to the E300 devices and the hsi1 interface
      #
      #**************************************************
    echo "Current VM Host ="$Current_VM_Host
      #*************************************************
      if [ "${Current_VM_Host}" = "$VM2" ]
        then
          echo "IFUP on hsi1 issued---> R91 Residency"
          sudo /sbin/ifup hsi1
          HSI_RC=0
        else
          echo "IFUP on hsi0 issued---> H91 Residency"
          sudo /sbin/ifup hsi0
          HSI_RC=0
      fi
  else
    echo
    echo "This Guest is not a valid LGR HSI candidate... "$Guest_id
    echo "   This script should not have been run on this guest."
    echo
fi
#
#-----------------------------------------------------
echo "exiting..."
echo "z/VM Linux Guest HiperSocketsInterface Reset Return Code is: " $HSI_RC
echo
#
# remove the control file to ensure integrity
rm /var/log/lxconusr/lgr_control_file
exit
#
```

6.3 Sample Zebra.conf for a HiperSockets guest (from LITDCON1)

see footnote 1 below

```
hostname xxxxxx.xxxxx.xxxxxx.ibm.com
password xxxxxx
enable password xxxxxxx
log file /var/log/quagga/quagga.log
!
interface dummy0
 ip address 10.20.7.100/32
 ipv6 nd suppress-ra
!
interface dummy0:0
 ipv6 nd suppress-ra
!
interface hsi0
 ip address 172.0.9.100/24
 ipv6 nd suppress-ra
!
```

```
interface hsi1
 ip address 172.1.2.100/24
 ipv6 nd suppress-ra
!!
interface trk1
 ip address 10.20.8.100/24
 ipv6 nd suppress-ra
!
interface trk2
 ip address 10.20.9.100/24
 ipv6 nd suppress-ra
!
interface lo
!
interface admn1
 ipv6 nd suppress-ra
!
interface htbt1
 ipv6 nd suppress-ra
!
interface sit0
 ipv6 nd suppress-ra
!
ip forwarding
```

6.4 Sample OSPFD.conf for a HiperSockets guest (from LITDCON1)

see footnote 1 below

```
interface dummy0
 ip ospf cost 1
 ip ospf priority 0
!
interface dummy0:0
 ip ospf cost 1
 ip ospf priority 0
!
interface hsi0
 ip ospf cost 1
 ip ospf priority 10
!
interface hsi1
 ip ospf cost 1
 ip ospf priority 10
!
interface trk1
 ip ospf cost 10
 ip ospf priority 0
!
interface trk2
```

```
 ip ospf cost 10
 ip ospf priority 0
!
interface lo
!
interface admn1
!
interface htbt1
!
interface sit0
!
router ospf
 ospf router-id 10.20.7.100
 network 10.20.7.100/32 area 4.4.4.4
 network 10.20.8.0/24 area 4.4.4.4
 network 10.20.9.0/24 area 4.4.4.4
 network 172.0.9.0/24 area 4.4.4.4
 network 172.1.2.0/24 area 4.4.4.4
 area 4.4.4.4 stub no-summary
!
password xxxxxx
enable password xxxxxx
log file /var/log/quagga/ospfd.log
```

[1]THESE MATERIALS ARE PROVIDED "AS IS" AND, SUBJECT TO ANY STATUTORY WARRANTIES WHICH CAN NOT BE EXCLUDED, IBM MAKES NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THESE SCRIPTS AND SAMPLE CONFIGURATION FILES. IBM WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THESE SCRIPTS OR THE SAMPLE OSPF AND ZEBRA CONFIGURATION FILES CONTAINED IN THIS PAPER'S APPENDIX.

ZSW03221-USEN-00