Linux Gen-Z Sub-system

Linux Plumbers Conference
Lisbon, Portugal
September 11, 2019

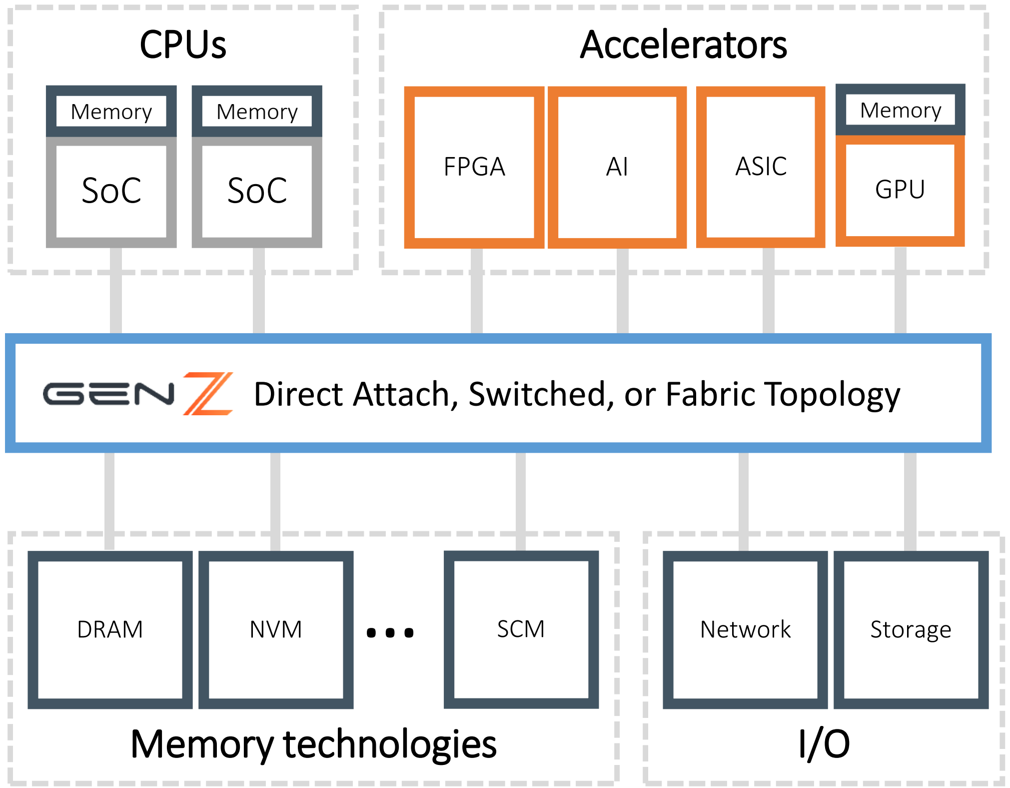
Jim Hull (jim.hull@hpe.com)
Betty Dall (betty.dall@hpe.com)

Agenda

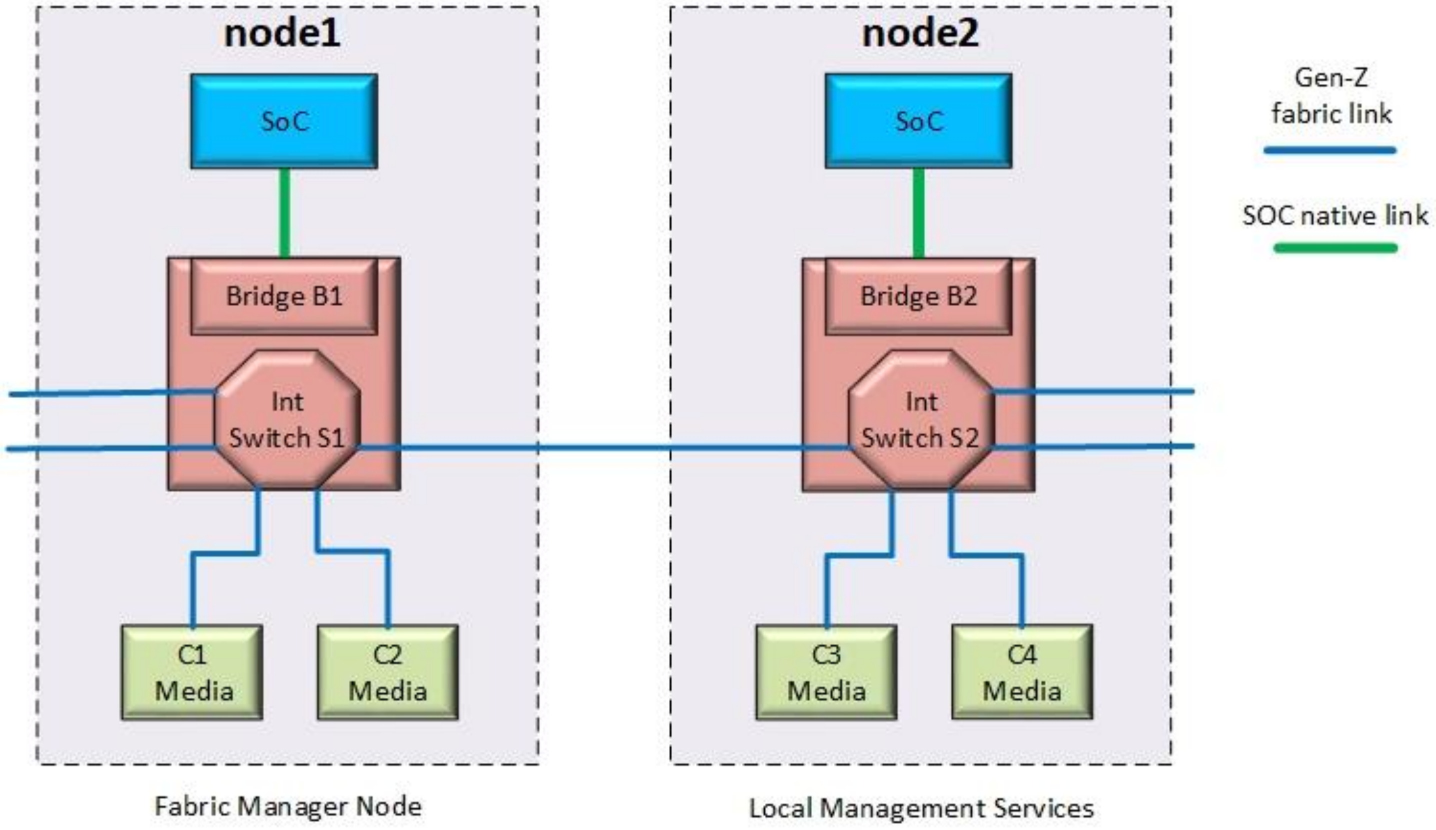
- Introduction to Gen-Z
- Kernel Sub-system
- Discovery
- Questions

Gen-Z, A New Open Interconnect Protocol

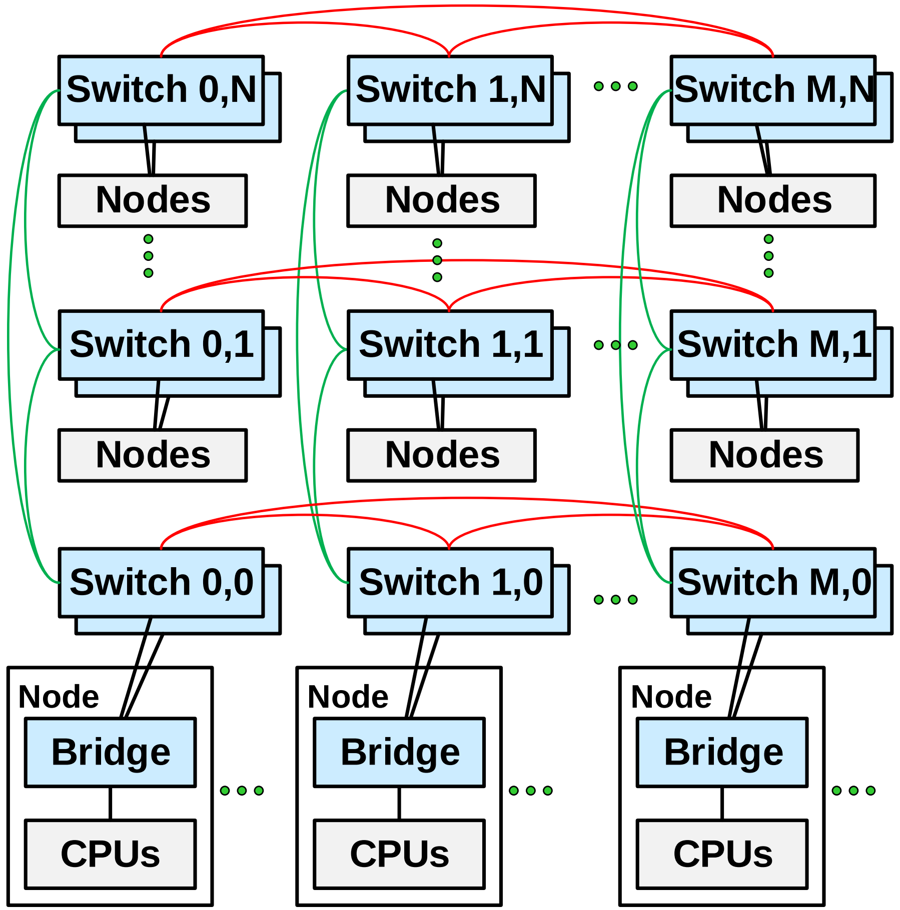
- Open consortium with broad industry support (70+ members)
- Family of Specifications: Core, Physical Layer, Mechanical, Scalable Connectors, Management
- Gen-Z is a memory semantic fabric that scales from 2 to 256M components
- PHY-independent protocol
  - Specific PHY determines latency/bandwidth/reach
  - 32 GT/s PCIe PHY, 25 Gbit and 50 Gbit 802.3 PHYs
- Can support an unmodified OS (e.g. firmware with ACPI support and Logical PCI Devices (LPDs))
- This talk is about modifying Linux for full Gen-Z support



Example Gen-Z Fabrics

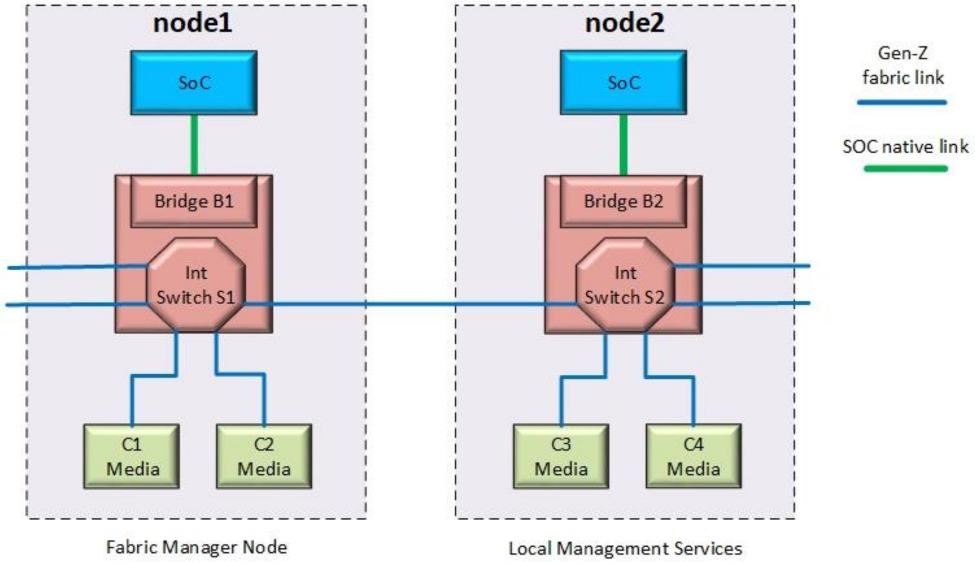


Simple 6 Component Topology



2D HyperX System Topology

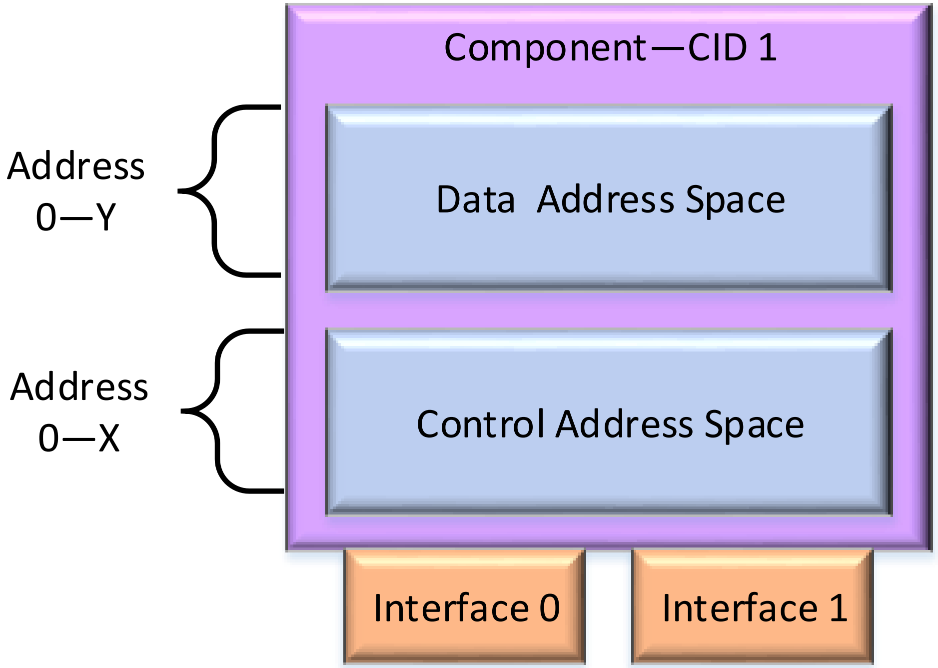
Gen-Z Management Software



- Gen-Z fabric spans multiple OS instances
  - No OS instance can assume it “owns” all components on fabric
- Components can be subdivided into resources
  - Example: a big media component split up
- A fabric manager assigns components/resources to each OS according to a “grand plan”
  - Describes components/resources using a DMTF Redfish specification
  - In-band vs out-of-band
  - Programs routing tables
- Local Management Services run on each OS instance
  - Consumes Redfish description for its OS instance

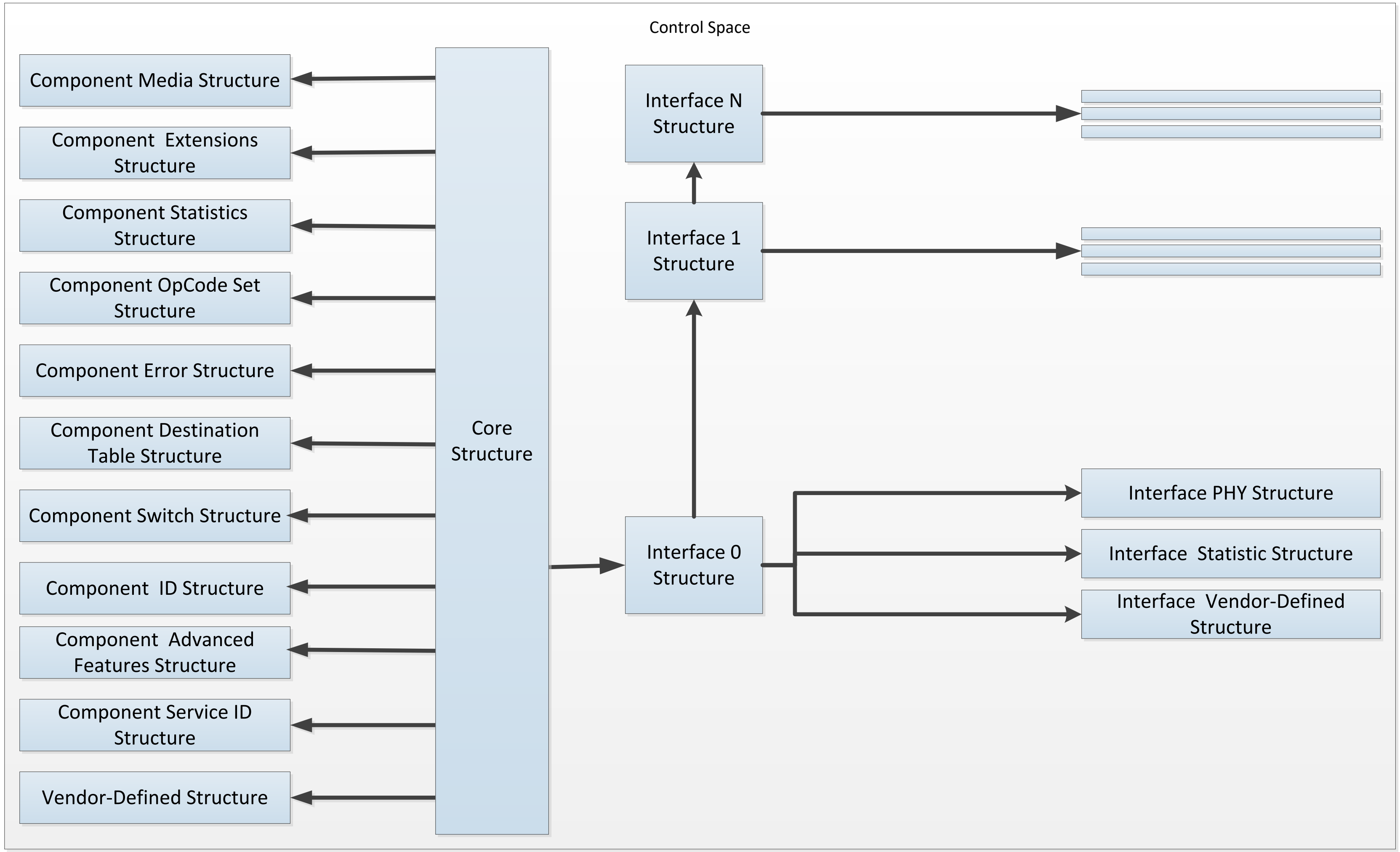
Basic Gen-Z Concepts

- Basic component roles
  - Requester: initiates packet
  - Responder: responds to request packet and sends acknowledgement (if specified)
  - Switch: routes packets from ingress interface to one or more egress interfaces
- Components have a 28-bit global component ID (GCID) assigned by management software
  - Optional 16-bit subnet ID (SID) plus 12-bit component ID (CID)
- Components have separate control and data space
  - Up to 2^52 bytes of control space for management
  - Up to 2^64 bytes of data space for component specific functionality
- Packets are unordered by default (big difference from PCIe)
- Software-managed coherence

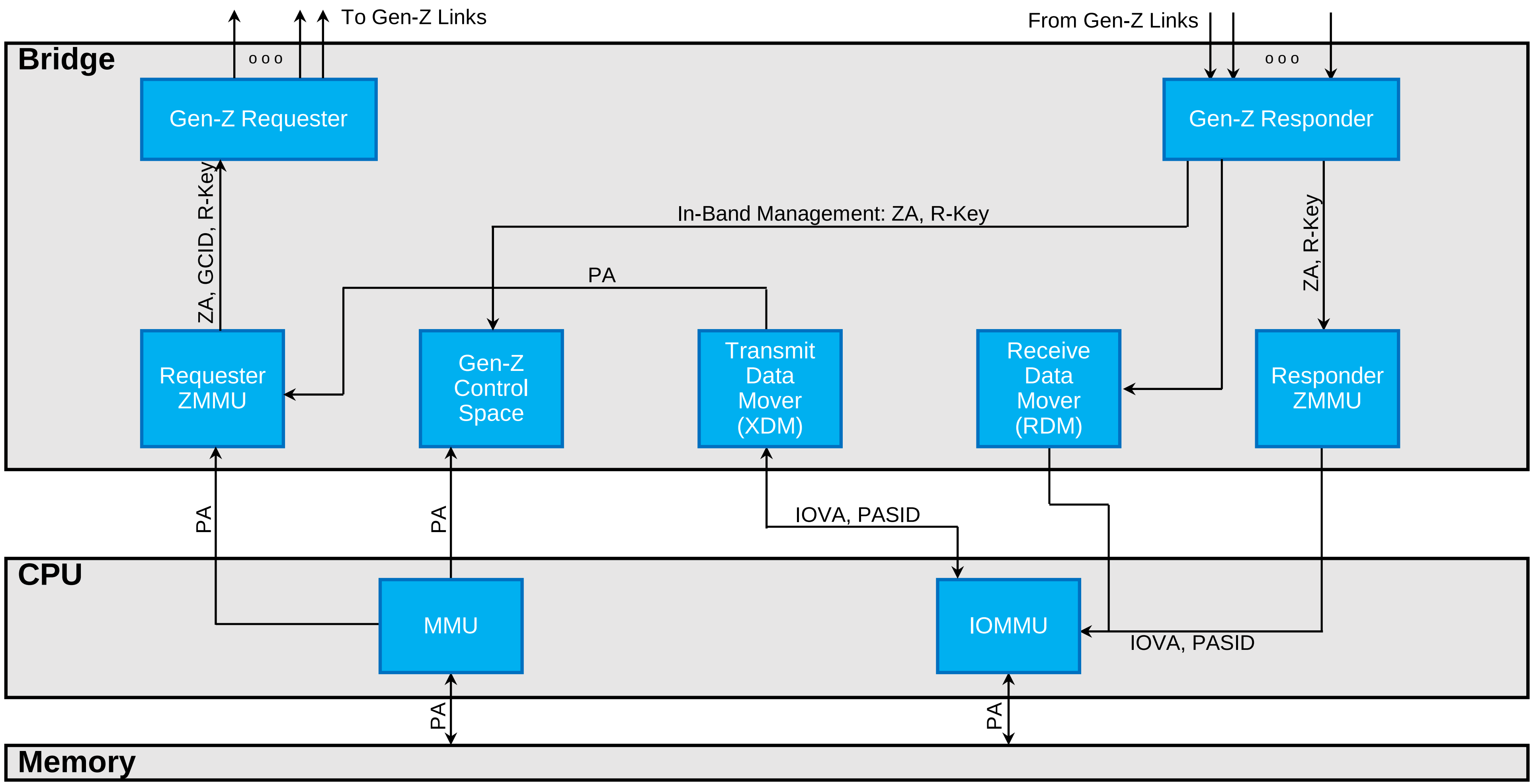


Control Space Structures

- Core Structure always at Control Space address 0
- Follow pointers to find other Structures and Tables



Bridge Component Block Diagram



ZMMUs

- OS-managed

- Requester ZMMU
  - Converts CPU/XDM physical address to Gen-Z address (ZA), checks PASID, and looks up GCID, R-Key, Traffic class

- Responder ZMMU
  - Data space only
  - Converts ZA to IOVA, checks the packet’s R-Key against PTE’s R-Keys, and looks up the PASID
  - IOVA and PASID passed on to IOMMU (if there is one), else PA passed on

- Page Grids vs. Page Tables
  - Page-Table-based ZMMUs have multi-level, forward-mapped page tables in local memory, with HW caching
  - Page-Grid-based ZMMUs have fixed number of PTEs on component, directly managed by OS

Agenda

- Introduction to Gen-Z
- Kernel Sub-system
- Discovery
- Questions

Why a Gen-Z Sub-system?

- Enable native device drivers, exposing the full capabilities of Gen-Z
  - Enables access to Gen-Z advanced features
  - Sharing of fabric resources across Linux instances
- Enable user space fabric managers and local management services
  - Both in-band and out-of-band fabric managers
- Why now?
  - So that Linux can support Gen-Z devices when hardware is available

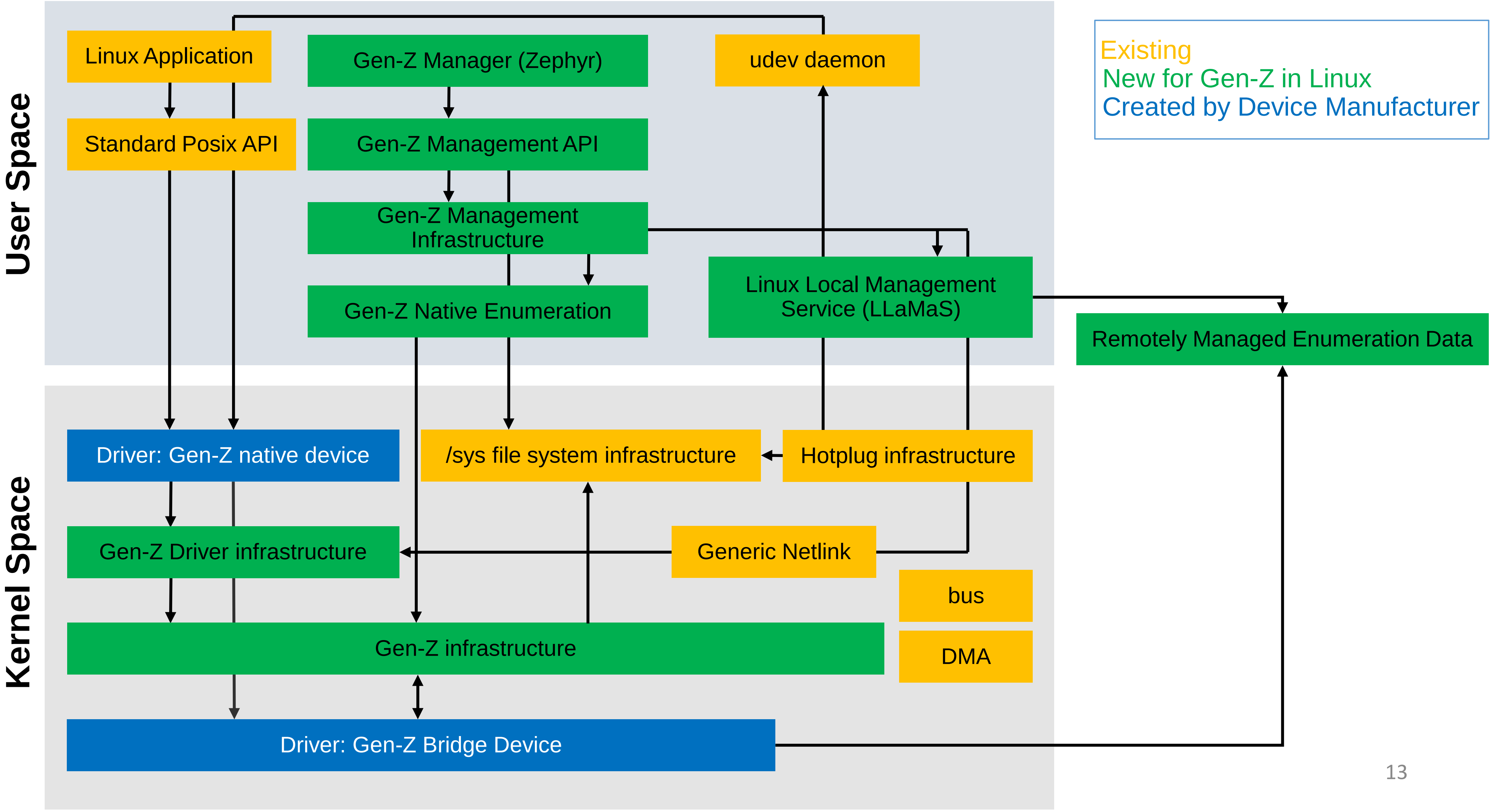
Gen-Z Advanced Features

- Interrupts
- Atomics
- R-Key Update Packets
- Buffer Requests
- Pattern Requests
- Multi-Op Requests
- Coherence Protocol
- Precision Time
- Lightweight Notification
- Wake Thread
- Packet Encapsulation
- Transparent Routers
- Strong Ordering Domains

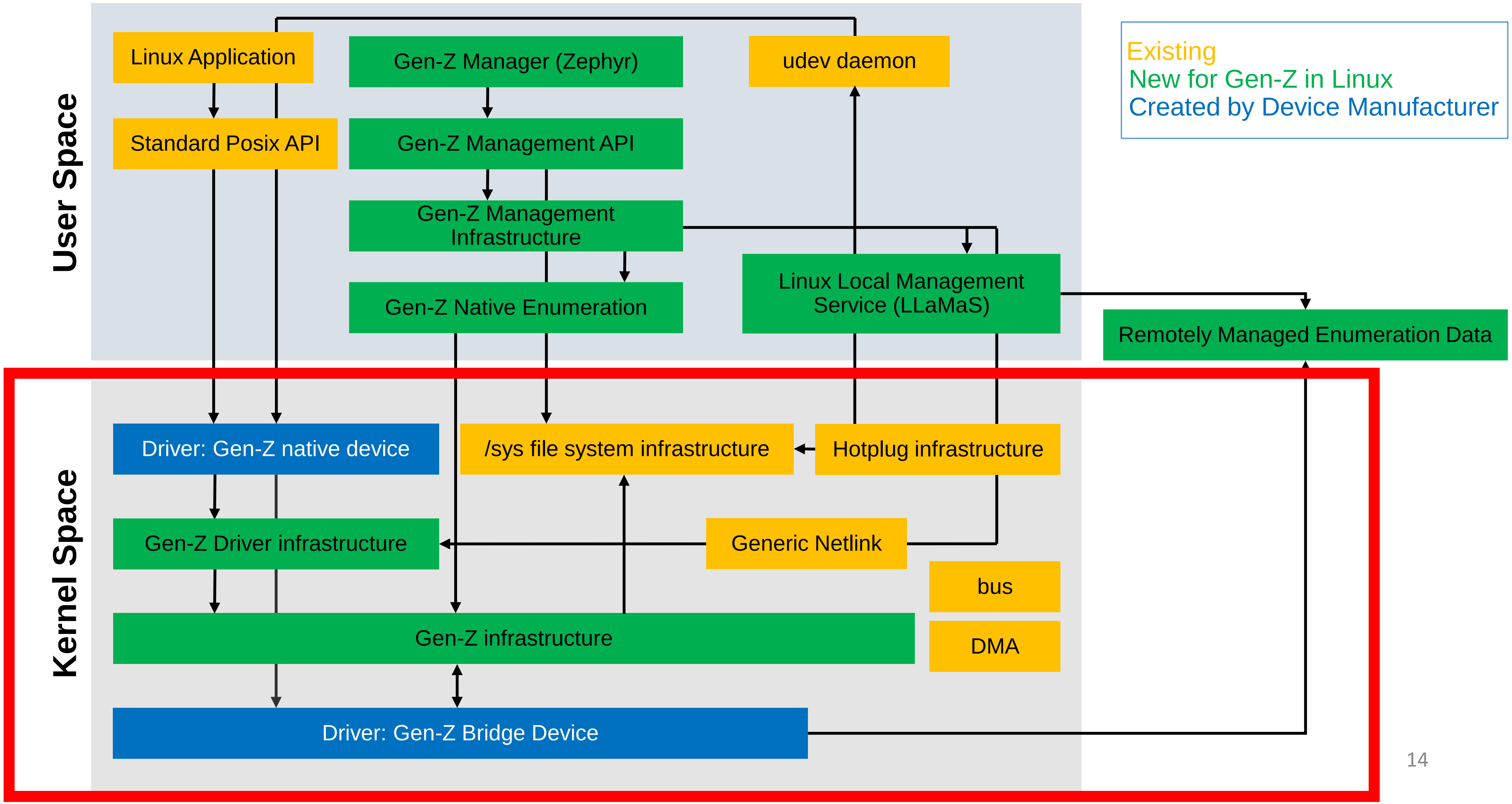
Design Considerations

- Be like PCI, USB and other existing buses when we can
- Policy in user space and mechanism in the kernel
- Use existing kernel services
- Deal with “almost everything is optional in Gen-Z”

Gen-Z Sub-system Block Diagram



Gen-Z Sub-system Kernel Interfaces



Bridge Driver Registration

- genz_register_bridge(struct device *dev, struct genz_bridge_driver *zbdrv);
  - Called during the driver probe function for the native bus of the bridge device driver.
  - Creates the sysfs file for the bridge device so that the Fabric Manager can start discovery
  - genz_bridge_driver structure has function pointers for:
    - Bridge info
    - Control space read/write/mmap
    - Data space read/write/mmap
    - Control write message
- genz_unregister_bridge(struct device *dev);

Device Driver Registration

- Similar to PCI's interfaces except driver matching is by UUID rather than vendor/device ID
- genz_register_driver(struct genz_driver *driver, struct module *mod, const char *mod_name)
  - genz_driver structure has function pointers for:
    - Probe
    - Remove
    - Suspend
    - Resume
- genz_unregister_driver(struct genz_driver *driver)

Sub-system ZMMU and IOMMU Management

- Map control space ZMMU entries for sysfs read/write
- Drivers map control/data resources through the ZMMU
- Still designing ZMMU API
  - Want to hide page grid vs. page table based ZMMU differences

- The Gen-Z sub-system needs to provide APIs for tracking PASIDs in the ZMMU and IOMMU
  - Question: Should there be a generic Linux interface for tracking PASIDs?

- Question: How do we map huge pages for Gen-Z device memory?
  - A Gen-Z Fabric can contain a large number of components each with an enormous data space
  - Gen-Z PTEs allow a choice of page sizes
  - For Page Grid based ZMMUs, there are a fixed number PTEs and so you have to use huge pages
  - Our understanding is that huge pages for device memory is not well supported

- Question: What is status of Shared Virtual Addressing (SVA) for the IOMMU?
  - The Gen-Z sub-system would use this proposed interface to hide IOMMU differences

Data Movers

- Kernel drivers like a block or eNIC driver would benefit from a generic data mover interface
  - Data mover queues can be assigned to other Gen-Z drivers
  - Drivers can use a data mover to generate Gen-Z packet types like atomics, write message, buffer and pattern requests
- RDMA drivers want to expose the native data mover hardware to user space
  - This argues for no generic Gen-Z sub-system data mover support
- Question: Should the Gen-Z sub-system implement a generic data mover interface?

Interrupts and Unsolicited Event Packets

- Not like PCI’s architected MSI/MSI-X interrupts
- Interrupt sources:
  - Gen-Z interrupt packets from components
  - Local bridge data movers
  - UEPs

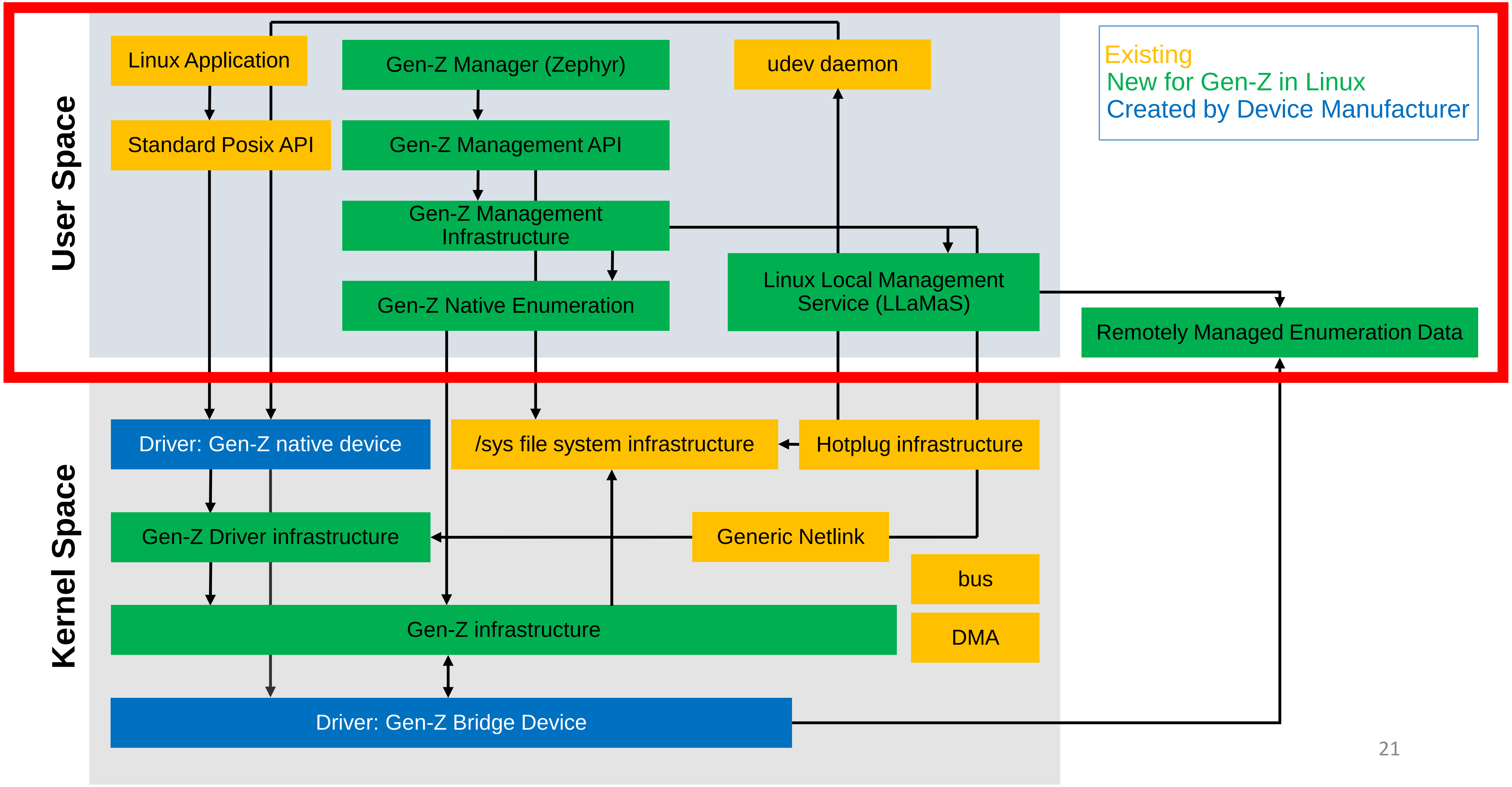
- Unsolicited Event Packets (UEP) signal fabric state changes like
  - Link-up/down
  - Hot add/remove of component
  - Errors

- UEPs become interrupts from the targeted bridge component
  - Vectored to sub-system and forwarded to user space

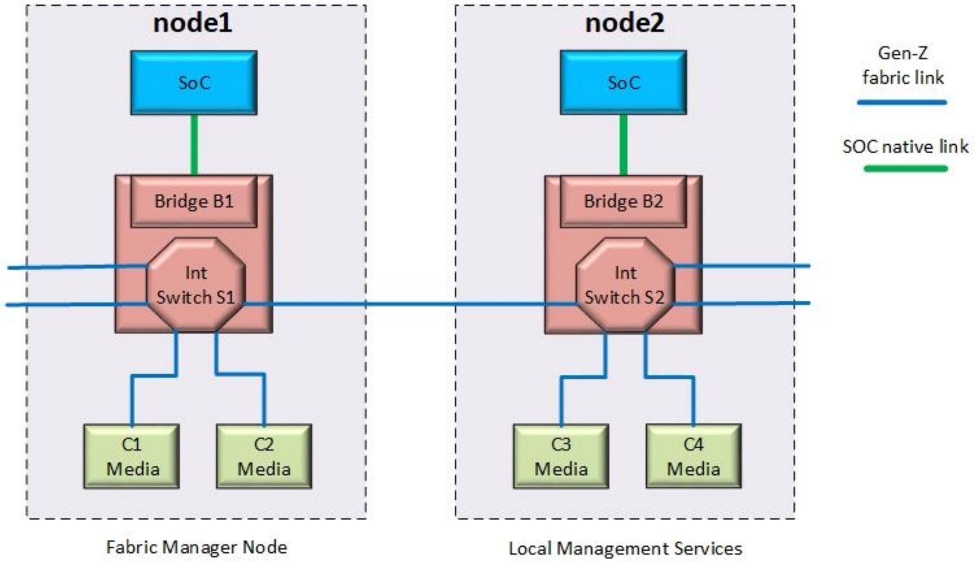
Agenda

- Introduction to Gen-Z
- Kernel Sub-system
- Discovery
- Questions

Gen-Z Sub-system User Space Interfaces



Gen-Z Discovery



- All nodes run local management services
  - For resources visible to a node, LLaMaS sends a Netlink “add component” command
  - Gen-Z sub-system creates a sysfs tree for resources in /sys/devices/genzN/SID/CID/RESOURCE
  - Gen-Z driver binds to resource’s UUID
- What Fabric Manager discovers: interfaces, switches, bridges, media
  - Fabric Manager does a recursive walk of the fabric to configure and assign GCIDs to all components
  - For all discovered components, Zephyr sends a Netlink “add fabric component” command
  - Gen-Z sub-system creates a sysfs tree for components in /sys/bus/genz/fabricN/SID/CID
- Generic Netlink communication to inform kernel of add/delete of components and resources
  - Question: Is generic Netlink the best choice for communication between user space and the kernel?

Managed Node sysfs Example

```
└── sys
    └── devices
        ├── genz0
        │   ├── 0000
        │   │   ├── 002
        │   │   │   ├── c_class
        │   │   │   ├── fru_uuid
        │   │   │   ├── gcid
        │   │   │   ├── memory0
        │   │   │   │   ├── control0
        │   │   │   │   ├── data0
        │   │   │   │   └── uuid
        │   │   │   └── memory1
        │   │   │       ├── control0
        │   │   │       ├── control1
        │   │   │       ├── data0
        │   │   │       └── uuid
        │   ├── bridge0 -> ../pci0000:42/0000:42:07.0/genz0
        │   └── mgr_uuid
        │
```

```
└── pci0000:42
           └── 0000:42:07.0
               └── genz0
                   ├── control
                   │   ├── component_c_access
                   │   ├── component_destination_table
                   │   ├── component_error_and_signal_event
                   │   ├── component_pa
                   │   ├── component_page_grid
                   │   ├── component_switch
                   │   ├── core
                   │   ├── interface
                   │   └── opcode_set
                   └── gcid
```

Fabric Manager Node sysfs Example

```
└── sys
    └── bus
        └── genz
            ├── devices
            │   ├── 0:0000:000 -> ../../../devices/genz/0000/000
            │   ├── 0:0000:002 -> ../../../devices/genz/0000/002
            │   └── 0:0000:003 -> ../../../devices/genz/0000/003
            └── fabric0
                └── 0000
                    ├── 000
                    │   ├── component_c_access
                    │   ├── component_destination_table
                    │   ├── component_error_and_signal_event
                    │   ├── component_pa
                    │   ├── component_page_grid
                    │   ├── component_switch
                    │   ├── core
                    │   ├── interface
                    │   └── opcode_set
                    ├── 001
                    │   ...
                    ├── 002
                    │   ├── component_c_access
                    │   ├── component_destination_table
                    │   ├── component_error_and_signal_event
                    │   ├── component_media
                    │   ├── core
                    │   ├── interface
                    │   └── opcode_set
                    ├── 003
                    │   ...
                    ├── 004
                    │   ...
                    └── 005
                        ...
```

Question: Is the proposed sysfs hierarchy consistent with Linux’s intended sysfs usage?

Agenda

- Introduction to Gen-Z
- Kernel Sub-system
- Discovery
- Questions

Summary of Questions

- Gen-Z uses PASIDs and the sub-system could use generic PASID interfaces. Any interest in this elsewhere in the kernel?
- How do we map huge pages for Gen-Z device memory?
- What is status of Shared Virtual Addressing (SVA) for the IOMMU?
- Should the Gen-Z sub-system implement a generic data mover interface?
- Is generic Netlink the best choice for communication between user space and the kernel?
- Is the proposed sysfs hierarchy consistent with Linux's intended sysfs usage?

References

- Gen-Z Consortium for specification: genzconsortium.org
- Gen-Z Linux Subsystem: github.com/linux-genz/linux
- LLaMaS github: github.com/linux-genz/llamas
- Alpaka github: github.com/linux-genz/python3-alpaka