DISTRIBUTED DEVICE BUS

A Thesis

by

AMRUTH KUMAR JUTURU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Riccardo Bettati
Committee Members,	A. L. Narasimha Reddy
	Jaakko Jarvi
Head of Department,	Dilma Da Silva

August 2015

Major Subject: Computer Science and Engineering

ABSTRACT

Peripheral devices are hardware components that are connected to a computer and they supplement the functionality of a computer. Over the years, a huge improvement has been observed both in variety and capabilities of peripheral devices. Starting from the input/output and storage devices of early days, today's peripheral devices support all aspects of a computer, with peripherals like Graphical Processing Units (GPUs) even supplementing the computational capabilities of a processor. At the same time, the support for peripheral devices in computers has vastly improved. While the earlier computers only supported static configuration of devices, the plug-and-play capabilities in present day computers allow devices to be added or removed at run time, thus reducing the complexity of managing peripheral devices. Today, it is not an exaggeration to state that, beyond the computational capability of a computer, it is the peripheral devices that define the user experience.

With the advancements in networking and distributed computing, the definition of what constitutes a computer has been blurred: Mainframes and Supercomputing clusters support batch processing, where processors/cores are treated as resources, and number of processors/cores available for a specific computation can be requested on demand. With cloud computing, users access services hosted across the Internet. However, usage models for peripheral devices have not caught up accordingly. For the most part, Peripheral devices are still limited to the computers they are physical attached to. Device virtualization solutions exist that can extend the device protocols over the network, enabling users to access devices connected to a different computer. However, these device virtualization solutions still need direct access to both the computer that has the device plugged in (Device Server) and to the computer that

intends to use the device (Device Client) and they do not support remote plug-and-play. So, there is a need for a device consolidation framework that supports new device usage models that are in line with the evolving models of computation.

In this thesis, we propose a framework called "Distributed Device Bus", which extends the concept of a conventional peripheral bus to include in its scope, the ports of all the computers that are connected over a network. Like a peripheral bus, a Distributed Device Bus is also associated with a computer called Master node. A Distributed Device Bus supports dynamic addition/deletion of ports and each of these ports can physically belong to any computer in the network. Computers that contribute ports to a Distributed Device Bus are called Provider nodes. A device plugged into any port that is assigned to a Distributed Device Bus is immediately made accessible to applications on master node. This device consolidation framework treats devices as a resource and access to a device is configurable rather than being limited to the computer the device is physically attached to.

DEDICATION

I dedicate my thesis to my parents Nagaraju Juturu and Krishna Veni.

ACKNOWLEDGMENTS

As the saying goes, *The mediocre teacher tells. The good teacher explains. The superior teacher demonstrates. The great teacher inspires*, Dr. Riccardo Bettati, the great teacher he is, continuously encouraged and inspired me throughout my thesis. For the guidance and support he has given me throughout the duration of my thesis, I express my deepest gratitude. Our discussions helped me improve my abstract thought process and build my technical skills.

Next I would like to thank my committee members, Dr. A. L. Narasimha Reddy and Dr. Jaakko Jarvi for their time and suggestions whenever I needed them. I am grateful for my committee members for letting me pursue my thesis under their auspices.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

With the advent of distributed systems, the concept of computation is being progressively decoupled from being associated with a single computer and is increasingly associated with a system of interconnected nodes that communicate and coordinate. This paradigm shift created a framework that handles computational power as a resource that is configurable, rather than being limited by hardware capabilities of a single independent computer. For example, in today's supercomputers and mainframes, it is not uncommon for a job to request hardware resources that are beyond the capabilities of a single computer. The requested resources are then assimilated by combining resources from a connected network of computers. For the duration of the job, these resources coordinate to fulfill the requirements of the job. At the end of the job, the resources are released and made available for rest of the system. Effectively, this paradigm supports the creation of an overlay of computational resources that are dynamically configurable, resulting in efficient usage of resources. This in turn gives rise to a scalable paradigm for computation.

Peripheral devices continue to be associated with a single computer. Conventional Peripheral Buses [1] limit the availability of a peripheral device to the single computer that happens to have the device physically plugged in. This static association of a peripheral device to a single computer is a limitation of conventional peripheral buses that prevents the possibility of treating a peripheral device as a resource that can configured to be accessed from any node in a distributed system. Even in today's supercomputers and mainframes, a computation that relies on a peripheral device is limited to the computer that has the device plugged in. No system is available to support the assimilation of devices attached to different computers in a network and

still provide a transparent device access to operating system services and user level applications. If an application relies on devices that are unavailable on the computing node hosting the application, there is no effective way to leverage available devices across the network.

Challenges posed by such a limitation on device access are four fold:

- Maintenance overhead: Each computer with that device attached has to be individually serviced whenever a particular device requires maintenance (such as update of device driver or reconfiguration of device parameters). Increasingly large number of sensor devices are deployed, typically attached to peripheral buses of host computers. For example, some sensor networks [2] consist of several computers and each computer is connected to one or more of sensor devices, each designed as a peripheral device. Computers in such networks can be distributed across wide areas. Maintaining such a network of devices by individually servicing each computer requires significant manpower and time.

- Security: Any system that accesses devices spread across wide areas exposes a sizable attack surface. A vulnerability related to a device can make any computer in the network a potential target. Compromising one such computer can put the entire network at risk. Consequently, each and every computer in the network should be constantly updated with security updates related to devices. Physical co-location of the host with the device causes various physical security vulnerabilities, which are particularly severe if the networked system is spread across wide areas.

- Inefficient usage of resources: Dependence of a computation on a device limits it to the computer with the attached device. This device dependent workload may prevent other workload from accessing the processor on the node that

has access to the device. This results in inefficient usage of computational resources, undue fragmentation of processor allocation and thus device dependent computations may have to wait, even if the present computation on the computer does not utilize the device. It is this coupling of resources (i.e., CPU and devices on the peripheral bus) that may unduly reduce utilization of the overall system.

- Scalability: Limitations on transparent device access limits the scalability of an application beyond the number of devices attached to the computer. In addition, this limitation restricts the usefulness of a device to the node in its physical location.

In order to overcome these limitations, a framework that supports consolidation of devices connected to different computers in a network is needed. This framework should not be limited to providing transparent device access but rather support a reconfigurable *Device Overlay*, a network of devices constructed on top of existing network of computers. Applications and operating system services can transparently leverage such a framework to scale out in terms of number of devices.

This document is organized as follows. In Section 2, we describe the criteria that guided the architecture of Distributed Device Bus. In Section 3, we describe the existing work in relevant area. In Section 4, we provide a brief model of Distributed Device Bus. Section 5 introduces a generic architecture of peripheral devices in operating systems, while Section 6 introduces a generic device virtualization architecture. Sections 7 and 8 describe architecture and design of our solution respectively. Section 9 provides implementation details. In section 10 we evaluate the system.

2. CRITERIA OF SOLUTION

Present day computers support a wide variety of device types, and each device type can have different protocols with varying capabilities. Peripheral devices differ in various aspects, including, but not limited to, transfer speed, bandwidth requirements, synchronous vs asynchronous operations, polled vs interrupt driven control, quality of service guarantees, error semantics. A framework that supports the consolidation of devices distributed across a network should not pose restrictions in terms of device access. Also, plug-and-play is one of the most important features of peripheral devices, and a device consolidation framework should extend plug-and-play functionality over the network. A good design for a device consolidation framework supports the following criteria:

2.1 Device Agnostic Service

Today's computers support a wide variety of devices. There are multiple standards that define the details of hardware cables, interfaces and protocols for data communications. For example, USB [3] is a standard developed for cables, connectors and communication protocols for data transfer between computers and devices. USB supports a wide variety of devices ranging from input/output devices, storage devices, cameras, speakers, microphones to game controllers. Similarly PCI [4], SCSI [5] are other bus standards that support a variety of devices. A device consolidation framework should support devices independent of the particular type of device or the type of peripheral bus.

2.2 Operating System Agnostic Service

A majority of the operating systems provide inherent support for all the popular bus specifications like USB, PCI and SCSI. The architecture of device consolidation framework should not impose any restrictions on Operating systems that the nodes in the connected network operate on.

2.3 Transparent and Complete Functionality

The proposed framework should support full functionality of the devices. Extending the device architecture over network should provide full functionality of the device, without necessitating any modifications to device drivers.

2.4 Plug-and-Play

One of the major features of current device architectures is plug-and-play. Plug-and-play enables computers to dynamically identify and support new devices without having to depend on a static device configuration. A proposed framework should extend the plug-and-play functionality across the device overlay.

2.5 Dynamic Reconfiguration

A framework to consolidate devices distributed across a network should not be limited by a static configuration. Each device in the network should be accessible to every other computer in the network and different computers access the device at different points of time. So, the association of a device to a computer in the network should be configurable, without affecting the state of other computers and devices in the network. Hence, the device overlay constructed on top of existing computer network should be reconfigurable.

In the following sections we describe a device consolidation framework that uses the device overlay paradigm to support transparent access of devices distributed

across the network. This framework extends plug-and-play over the network and is independent of device type and operating system. Access to devices in this framework is configurable on demand.

3. CURRENT WORK

A variety of device virtualization solutions have been developed over the years that extend the device protocols over networks, and that enable transparent access of remote devices. We observe, however, that such virtualization solutions are specific to either the type of device or to operating system, often to both. Also, purely device-virtualization based solutions do not support plug-and-play and reconfigurable device overlays.

3.1 iPCI

Peripheral Component Interconnect (PCI) [4] is a local bus standard that supports hardware devices to be plugged into the computer. iPCI [6] is a protocol extension of PCI over networks. iPCI proposes to encapsulate PCI transport packets into network packets, thus enabling the communication between the bus and device over a network. This makes iPCI to support any kind of PCI device. However, this protocol has been described in the literature only, and we are not aware of a working implementation.

3.2 Modbus TCP/IP

The Modbus protocol [7] proposes a master/slave architecture among the devices connected to a serial bus. Requests are always initiated by the master, and slaves accept requests and process them. Modbus TCP/IP [8] is an extension of Modbus over TCP/IP. The protocol is independent of the particular device, the application or the operating system. Because it is using a master/slave architecture, the protocol does not allow for reconfiguration or device discovery.

3.3 USBIP

USBIP [9] is an extension of USB [3] over network. USB Request Blocks (URBs) [10] are USB protocol units that carry the device driver's request for the device. In USBIP, URBs are transported over the network, enabling communication of device driver on one machine (Device Client) with the device on another machine (Device Server). Since URB is a generic structure that is independent of the type of USB device and operating system, USBIP is independent of the type of USB device. It therefore supports full functionality of the remote USB device without requiring any changes to device drivers. Since applications use the same device driver for accessing both a local and remote USB device, applications are unaware if they are accessing local or remote device. Hence, USBIP provides support for transparent access of remote USB device.

3.4 Remote USB Ports

Protocol remoting solutions like USBIP deal with devices, necessitating the physical presence of device for the solution to work. Such solutions do not support plug-and-play for remote devices. Remote USB Ports [11] provide an abstraction of USB ports and support plug-and-play for remote devices. In addition, this architecture supports user authentication and secure transfer of URBs over the network. However, each remote port has to be maintained manually, and assimilation of devices connected to different nodes in a network becomes cumbersome.

3.5 iSCSI

Small Computer System Interface (SCSI) [5] is a peripheral device interface standard that defines physical interface, commands and protocols. SCSI can support a wide variety of devices. However, the device types that most commonly use SCSI

are hard disks, CD drives and scanners. Internet Small Computer System Interface (iSCSI) [12] is a Storage Area Network protocol that extends SCSI over IP networks.

3.6 Software Buses

The Software Bus [13] paradigm is widely used in software architecture to develop a platform that connects software modules, components or objects over a shared communication channel. Examples of software buses range from middle ware systems such as CORBA [14] to bus implementations such as Ivy [15], SWBus [16], iBus [17], Toolbus [18] and Microsoft's Enterprise Service Bus (ESB) [19].

While Software Bus architectures support the design and implementation of loosely coupled components that are distributed across a network and coordinate by communicating via messages, there is no direct equivalent in literature for a device consolidation framework described above. However, such a device consolidation framework borrows many of the design principles proposed in various bus architectures.

4. DISTRIBUTED DEVICE BUS - A USECASE

We propose an architecture for a device consolidation framework called *Distributed Device Bus* (DDB). Like a peripheral bus, a DDB belongs to a single computer, which we call *Master node*. What distinguishes a DDB from a traditional peripheral bus, however, is the ability to have ports that can physically belong to any participating computer (called *Provider node*) in the network. A device plugged into any of the assigned ports of a DDB becomes accessible to Master node. Ports can be dynamically assigned to/removed from a DDB. Many such DDBs can be defined over the network. A conceptual illustration of a DDB is presented in Figure 4.1.

In Figure 4.1, Node 0 is the master node for DDB *DBUS1* and Port 1 on Node 1 and Port 3 on Node 2 are assigned to DBUS1. Node 2 is the master node for DDB *DBUS2* and Port 3 on Node 1 is assigned to DBUS2.

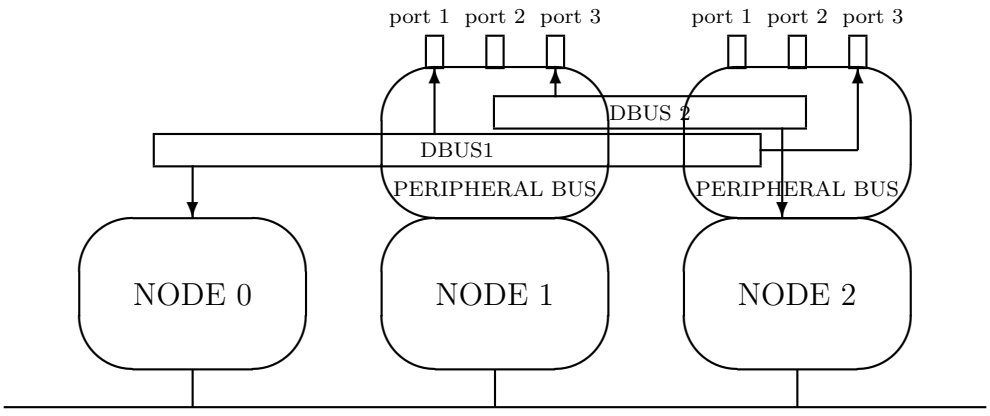

Figure 4.1: An Illustration of Distributed Device Bus

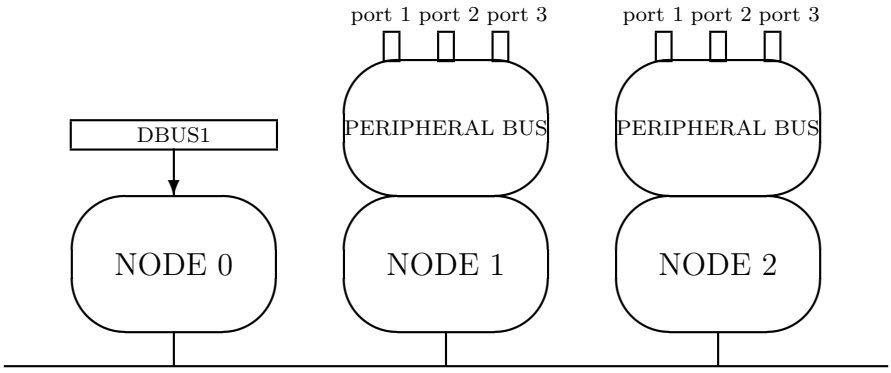

Figure 4.2: Creation of a Distributed Device Bus

For the sake of simplicity, pictures in Section 4 depict Node 0 with out its own peripheral bus. In reality, every participating node in the Distributed Device Bus infrastructure can have its own peripheral buses. The solid lines connecting different nodes in the figures represent the network.

In the following sections, we take an operational approach to define the DDB concept, and we describe the operations supported by the DDB:

1. Create a Distributed Device Bus

   Figure 4.2 illustrates a DDB after creation. Node 0 is the master node for DDB DBUS1. With no ports assigned to it, the scope of DBUS1 is limited to Node 0. A DDB with no ports assigned to it provides no functionality other than book keeping.

2. Add a Port to Distributed Device Bus

   Hardware ports belonging to any participating computer in the network can be assigned to a DDB. When a port is assigned to a DDB, the DDB is said to have been extended over network. The computer that provides the port to

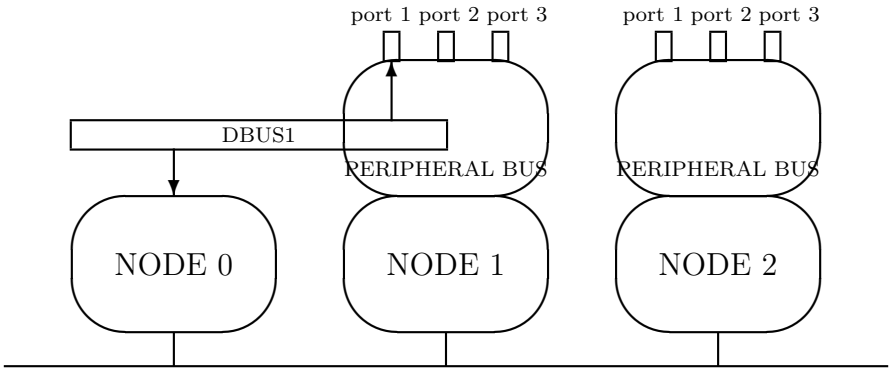

Figure 4.3: Adding a Port to Distributed Device Bus

a DDB is called *Provider node* and the scope of DDB now includes the new provider node.

Figure 4.3 illustrates adding a port to a DDB. Here, Port 1 on Node 1 is added to DDB DBUS1. Hence Node 1 acts as Provider node for DBUS1. As illustrated in the figure, this results in extension of DBUS1 across the underlying network to Node 1.

Figure 4.4 illustrates adding Port 1 on Node 2 to DBUS1. In this figure, DBUS1 expands further over the network, and Node 1 and Node 2 act as provider nodes and Node 0 remains the Master Node.

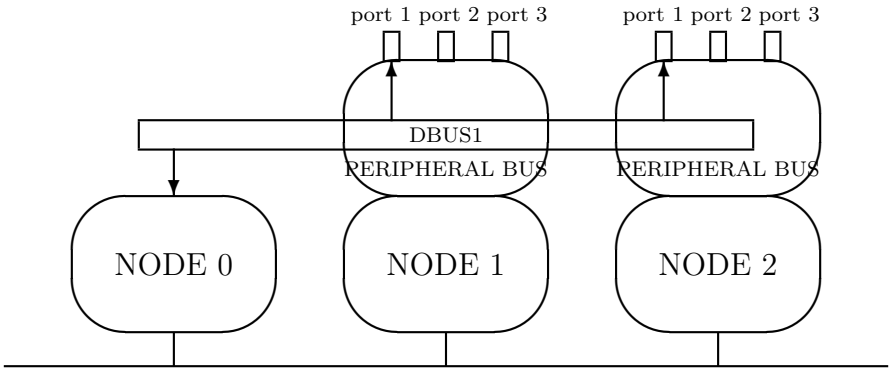

Figure 4.4: A Distributed Device Bus

3. Delete a Port from a Distributed Device Bus

   Ports assigned to a DDB can be dynamically deleted, resulting in reduction of scope for the DDB. A device attached to deleted port will no longer be accessible from the Master node.

   Figure 4.5 illustrates the deletion of Port 1 on Node 2 from DBUS1. This results in reduction of scope for DBUS1 and Port 1 on Node 2 becomes accessible to Node 2.

4. Delete a Distributed Device Bus

   Deleting a DDB results in deletion of all port assignments of the DDB. Ports on provider nodes that are previously assigned to DDB become locally accessible.

   Figure 4.6 illustrates deletion of DBUS1. Consequently, Port 1 on Node 1 becomes accessible to Node 1.

In summary, we observe that DDB's support the same operational semantics of a traditional peripheral buses, including their plug-and-play capabilities. DDB's

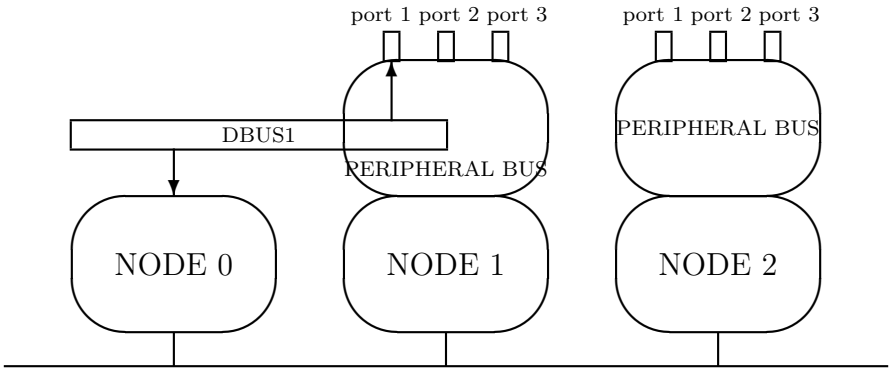

Figure 4.5: Deletion of Port from Distributed Device Bus

support a variable number of ports however, and each port can belong to different computers in the connected network. In addition, ports can be dynamically added/removed from the distributed device bus. The DDB's naturally extend plug-and-play capabilities to across the network.

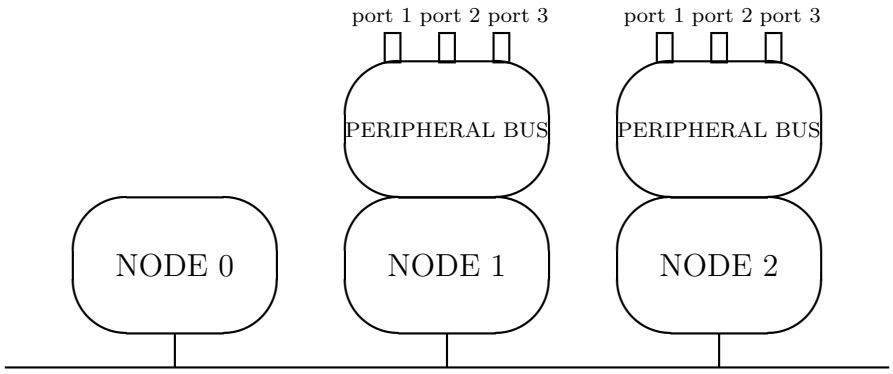

Figure 4.6: Deleting a Distributed Device Bus

5. TRADITIONAL PERIPHERAL DEVICE ARCHITECTURE

The Distributed Device Bus architecture is a natural extension of the traditional peripheral device bus architecture. We will therefore use the peripheral device architecture as a starting point. Although specific details may vary across device types, the device architecture is essentially defined by the specific peripheral bus and is therefore largely similar across the devices. The idea of Distribute Device Bus leverages these similarities to build a device agnostic bus architecture.

5.1 Peripheral Bus

A peripheral bus is a standard that describes the details of hardware (wires, optical cables and other electric components), software components, and communication protocols followed for exchange of data between computer and devices. Even though a peripheral bus is a single logical abstract entity in itself, the actual functionality as specified by the standard is manifested in operating systems by communication and cooperation between multiple hardware and software entities. Again, differences may exist between the complexities and functionalities of these components that realize the specifications of different standards. Sometimes differences exist even between different versions of the same standard. A typical peripheral bus consists of the following components:

1. Host Controller

   A host controller is a hardware component that provides the connection between the computer and peripheral devices. Peripheral devices are plugged into the hardware ports provided by the host controller. Essentially, data communication between a computer and the peripheral device happens via the host controller.

2. Host Controller Driver

   A host controller driver is the operating system driver that manages the host controller. Besides managing the interactions at the hardware level, the primary responsibility of the host controller driver includes the implementation of communication protocols as specified in the bus standard. However, the entire data communication protocol is not necessarily implemented in the host controller driver. Different host controller drivers provide different levels of functionality to support the data communication protocols between the computer and peripheral devices. The host controller driver provides the standard interface for device drivers to communicate with the device.

3. Host Controller Interface

   A Host Controller Interface specifies the register-level interface that facilitates the communication between host controller driver and host controller. The host controller interface determines the complexity of the host controller driver to realize the specification of the bus.

The generic idea of a peripheral bus includes all of the above three components. The Figure 5.1 illustrates the generic concept of a peripheral bus. Subsequent usage of the term "peripheral bus" refers to the combination of the above mentioned components.

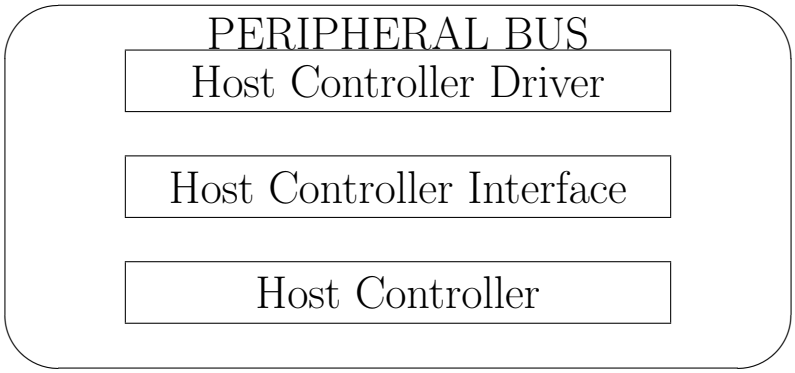

Figure 5.1: Generic Peripheral Bus

5.2 Peripheral Device Architecture

A peripheral device is plugged into the hardware port provided by the peripheral bus. The peripheral bus, through the functionality provided the coordination of *host controller, host controller interface* and *host controller driver*, loads a device driver matching the requirements of the peripheral device plugged into the hardware port. A device driver provides the standard interface for applications running on a computer to access the device.

Figure 5.2 illustrates the generic architecture for peripheral device in a computer. Even though differences may exist between the device driver architectures of different operating systems, Figure 5.2 still captures the essence.

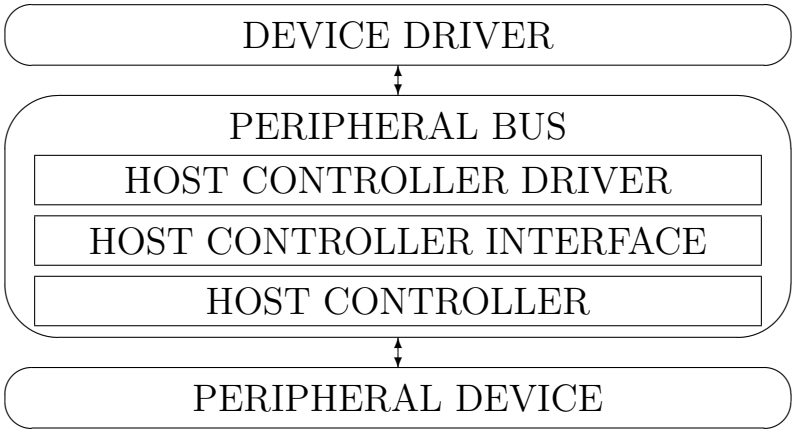

Figure 5.2: Generic Device Architecture

6. GENERIC DEVICE VIRTUALIZATION ARCHITECTURE

Our implementation of DDB leverages existing device virtualization solutions. Hence this section is dedicated for describing a generic device virtualization architecture. A device virtualization solution enables a computer to access a remote device (a device connected to a different computer in the network). The computer that has the device physically plugged in is called the *Device Server* and the computer that accesses the device is called *Device Client*.

As explained in Section 5.2, a Peripheral Bus captures the device driver's request for the device and provides the functionality to communicate with the device and get the device driver's request serviced. A device virtualization solution that supports transparent access of a remote device does not necessitate any changes in device driver. Such a device virtualization solution should distribute the responsibility of a peripheral bus as explained in Section 5.1 between the device client and device server, which is achieved by the communication and coordination of the following components

6.1 Device Server Proxy Module (DSPM)

A DSPM is responsible for hiding the device from applications on device server and setting up an infrastructure that enables it to receive device driver's request for the device from device client, service these requests by accessing the device and communicating the response back to the device client. Once a device to virtualize is identified, a DSPM hides the device from applications on the device server by preventing the operating system mechanism that sets up infrastructure as explained in Section 5.1. Instead, DSPM sets up the infrastructure that receives device requests from device client, services those requests using the device and reports the results to

device client. Once this infrastructure is setup, the device is said to be *EXPORTED*.

6.2 Device Client Proxy Module (DCPM)

A DCPM is responsible for loading a device driver suitable for the remote device, capture this device driver's request for the device and delegate the processing of these requests to device server. Once a device is exported as explained in the Section 6.1, DCPM communicates with the DSPM to get the details of the device and leverages the operating system functionality to load the appropriate device driver for the device and setups the infrastructure responsible for delegating the device request processing to device server. Once this infrastructure is setup, applications on device client will be capable of accessing the remote device using the standard operating system interface and the device is said to be *IMPORTED*

The Figure 6.1 illustrates the generic device virtualization architecture.

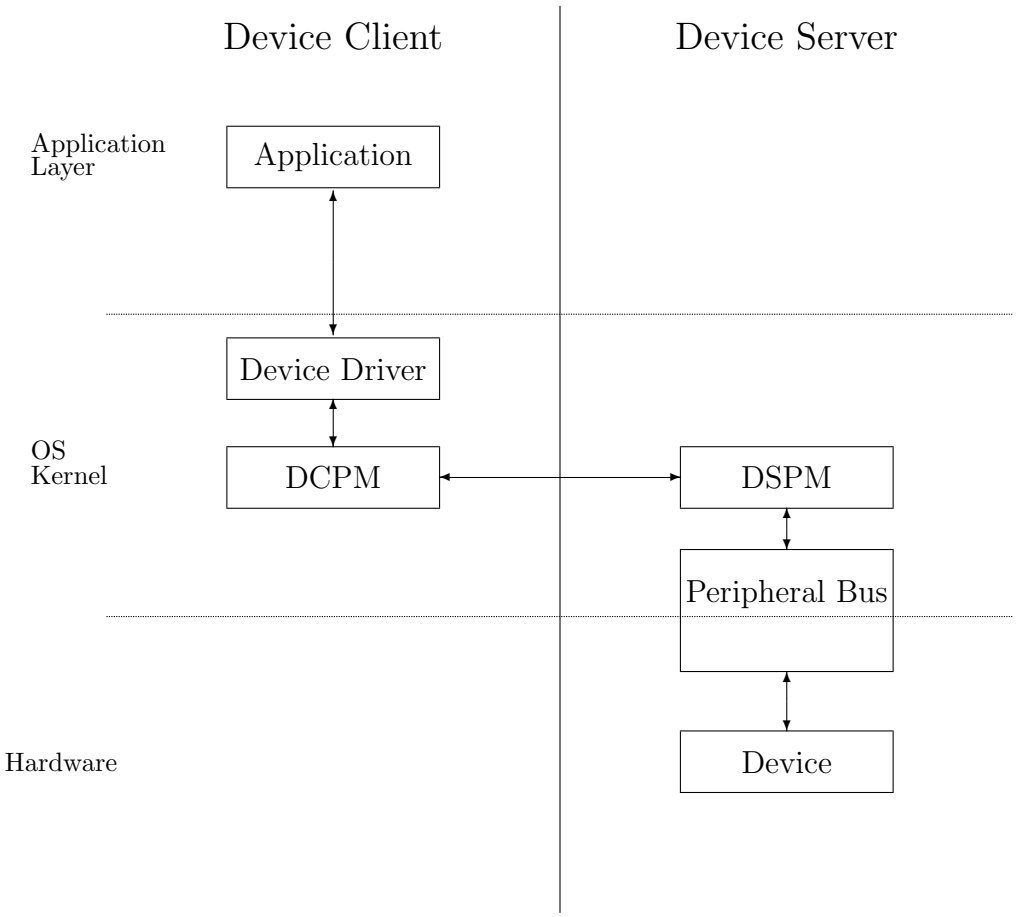

Figure 6.1: Generic Device Virtualization Architecture

7. DISTRIBUTED DEVICE BUS ARCHITECTURE

The architecture of the Distributed Device Bus follows a layered model, where lower layers provide communication and virtualization services to higher layers. Specifically we identify four layers, which we call *Device Virtualization Layer, Event Management Layer, Service Matching Layer* and *DDB Management Layer*. In the following sections we describe the rationale for and the services provided by each layer.

7.1 Device Virtualization Layer

The Device Virtualization Layer provides the communication services needed for the master node on the DDB to remotely access and control devices across the network. Typical services provided at this level are tunneling of the communication protocol across the network and device proxy services to enable transparent access of the remote device from device drivers on the master node. This enables applications on master node to transparently access device on the provider node.

As we described in Section 3, a number of approaches for device virtualization exist. We described in Section 6, how many of these systems follow a generic architecture, which in turn can be easily leveraged as device virtualization layer for DDB.

Our implementation of DDB can therefore make use of existing device virtualization systems such as USBIP [9] and Remote USB Ports [11] in Device Virtualization Layer, and we describe in the Section 8 and Section 9, how we use USBIP in the Device Virtualization Layer.

7.2 Event Management Layer

Hardware events related to devices define the operational semantics of devices on a computer. The event of a device being plugged in, for example, triggers specific operating system mechanism that loads the device driver for the device, thus enabling application access to the device. Similarly, whenever a device is removed, this triggers operating system mechanism to unload the device driver for the device, thus terminating the device access for applications. In a DDB, where applications on the master node access both local and remote devices, the hardware events related to a device must be transported over the network. The Event Management Layer provides the required functionality.

The Event Management Layer manages and responds to device configuration events, such as when a new device is plugged-in or removed from a port. Essentially, the Event Management Layer sets up the Device Virtualization layer so that device proxy and tunneling functionality in Device Virtualization Layer can be initiated or terminated accordingly.

7.3 Service Matching Layer

The DDB model is expected to reflect the operation of a local peripheral bus. Once the device virtualization layer has set up the connection between master node and the remote device, and the appropriate device drivers are loaded, the functioning of the device is identical to that of a device on a local peripheral bus. Some of the device virtualization mechanisms may not be able to transparently enable plug-and-play. However, in some cases, for example Modbus [8] the remote device must be manually configured to act as a remote device. In others, for example USBIP [9], the device virtualization system makes available remote device, and not remote ports. Plug-and-play, therefore, is not directly supported. The Service Matching

Layer compensates for the inconsistencies between the device model supported by the Device Virtualization Layer and the device model required by the Distributed Device Bus. If the functionality supported by Device Virtualization Layer cannot support remote ports as required by Distributed Device Bus, for example, the Service Matching Layer provides appropriate functionality to compensate for the difference in requirements.

7.4 Distributed Device Bus Management Layer

The Distributed Device Bus Management Layer provides the functionality to store and manipulate the state of Distributed Device Buses defined in the system. This information includes the Distributed Device Buses defined over the network along with the information of ports assigned to the Distributed Device Buses. This layer also provides necessary tools that enable users to interact with Distributed Device Buses in a network. Finally it implements the interface required for creating, manipulating and deleting Distributed Device Buses in the network.

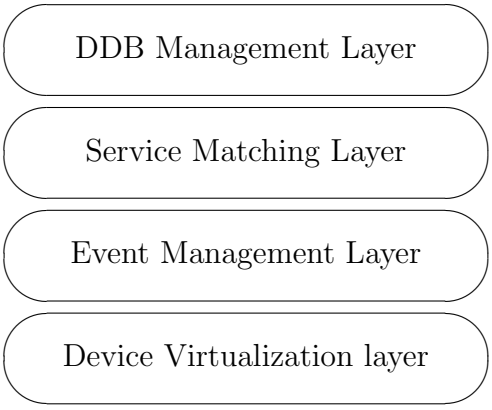

Figure 7.1: Distributed Device Bus Architecture

8. DESIGN

Besides fulfilling the criteria explained in Section 4, the design of distributed device bus infrastructure is influenced by the following requirements:

1. Lightweight

   A distributed device bus provides an infrastructure for assimilating devices present in all nodes in a connected network. Hence the overhead of participation on each node should be minimal. Each node in the network might have other responsibilities besides its participation in distributed device bus infrastructure. Hence the software modules running on each node should be lightweight.

2. Minimal Network Traffic

   As explained above, participation in the distributed device bus infrastructure may not be the only responsibility of the node and hence, even the interconnection medium for the nodes might as well be used for other communication purposes besides the communication required for device assimilation framework. As a result, it is important to reduce the amount of traffic through the interconnection medium.

3. Compatibility with existing Device Virtualization Mechanisms

   Our DDB design must be able to leverage existing device virtualization systems, which as described in Section 6 typically rely on proxying and tunneling of device protocols. In some cases, there is a mismatch between the services of the DDB and those provided by a particular device virtualization system.

   For example, as described in Section 3.3, USBIP virtualizes USB devices, thus necessitating the presence of the device for the proxy mechanism to work.

However, our DDB model extends the USB bus across the network providing the support for remote ports. Appropriate mechanisms must be put in place to resolve such mismatches.

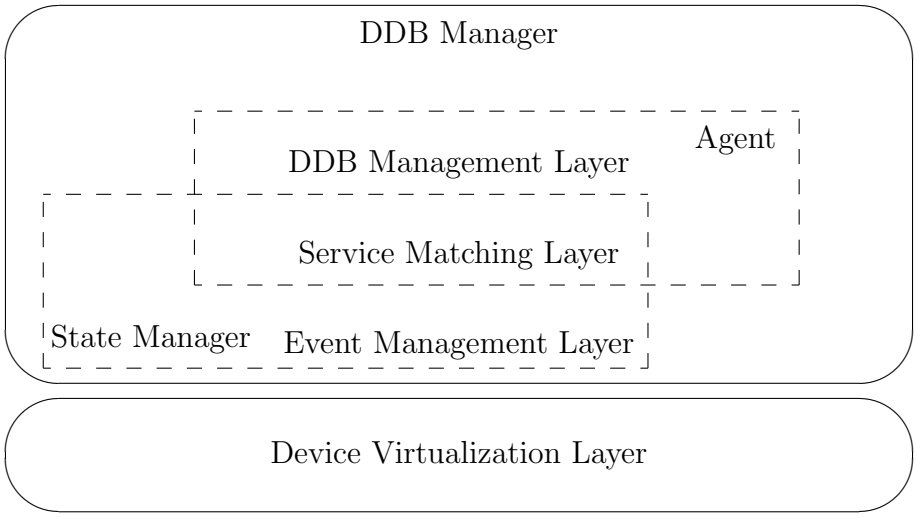

Figure 8.1: Distributed Device Bus Design

As a proof of concept for the Distributed Device Bus, we implemented a Distributed Device Bus for USB devices that assimilates USB devices distributed over the network. The following sections describe how the functionality of different layers of Distributed Device Bus as explained in Section 7 are realized in our implementation.

8.1 Device Virtualization Layer

Our implementation of Device Virtualization Layer utilizes USBIP [9], a USB device virtualization solution, that supports remote USB device access. Figure 8.2

describes the design of USBIP.

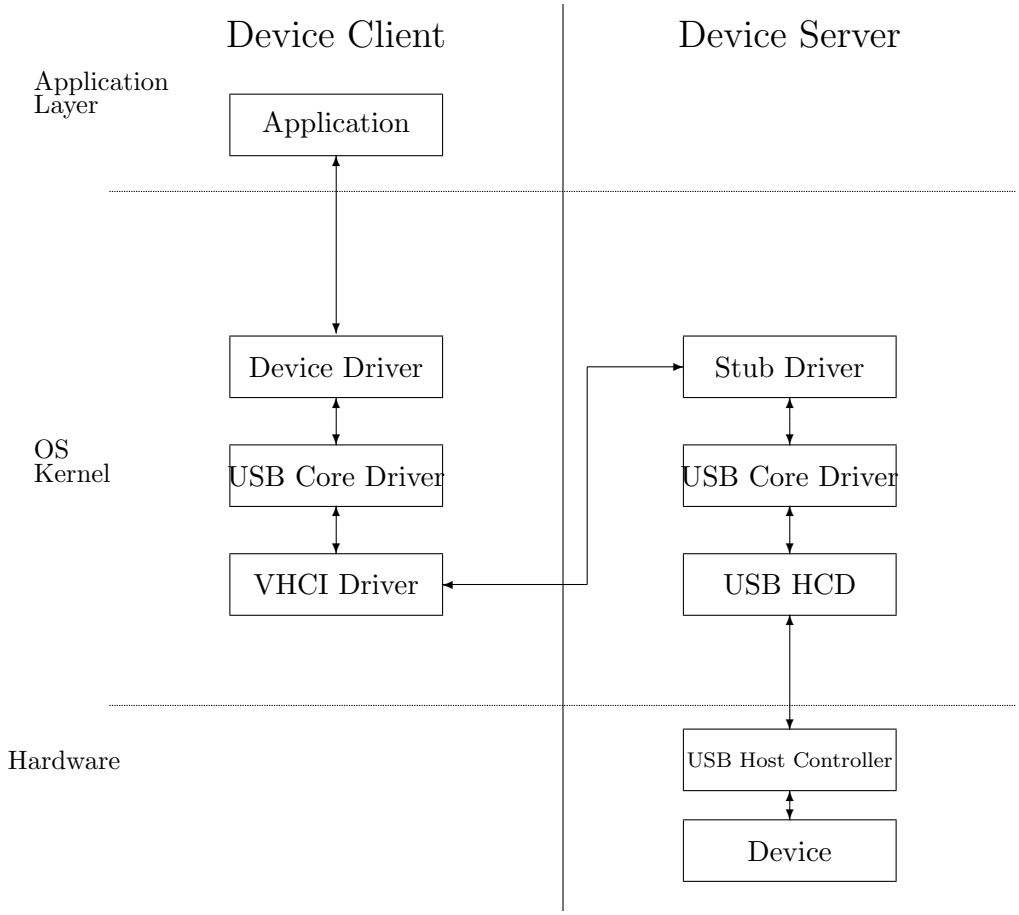

Figure 8.2: USBIP Design

Virtual Host Controller Interface Driver (VHCI Driver) and Stub driver are the kernel modules introduced by USBIP.

8.1.1 Stub Driver

Stub Driver, in coordination with the USB core driver and USB Host Controller Driver (USB HCD) on the device server, implement the functionality of the Device Server Proxy Module as explained in Section 6.1. The stub driver on device server

hides the device from applications on device server and is responsible for receiving device driver requests for the usb device, servicing the requests by accessing the usb device and reporting the results to Device Client.

8.1.2 Virtual Host Controller Interface Driver

The Virtual Host Controller Interface Driver, in coordination with the USB core driver is responsible for implementing the functionality of the Device Client Proxy Module as explained in Section 6.2. VHCI Driver communicates with Stub driver on device server to gather the USB device information required to load the corresponding USB device driver on device client. Also, the VHCI driver is responsible for capturing the device driver's requests for USB device and delegate the processing of device requests to stub driver.

8.2 Distributed Device Bus Manager

The Distributed Device Bus Manager realizes the combined functionality of Event Management Layer, Service Matching Layer and Distributed Device Bus Management Layer as explained in Sections 7.2, 7.3 and 7.4 respectively. Distributed Device Bus Manager is composed of the following components.

8.2.1 State Manager

For a conventional peripheral bus, information about the devices available in the system are stored with in a single computer, which is responsible for managing device related hardware events like device plug-in and plug-out events, defining the operational semantics for devices in a single system. However, the potential scope of a DDB includes the whole of the connected network. Hence, information of the DDB should be accessible to every participating node in the network. To avoid replicating the data on each participating node, a State Manager, which is accessible

from every participating node in the network is used to store the information about DDBs defined over the network.

We implemented the State Manager as a data store using Redis [20]. Redis is a open source key-value data store, that is used to store information about all the DDBs. This Redis data store is accessible from all the participating nodes in our implementation. Also, Redis supports notifications. Updates to status of ports assigned to DDBs are notified to relevant nodes in the network. This notification mechanism is leveraged to contribute to the functionality of Event Management Layer as explained in Section 7.2, Service Matching Layer as explained in Section 7.3 and Distributed Device Bus Management Layer as explained in Section 7.4

8.2.2 Agent

A computer in the network can either be a Master node or Provider node or both with respect to different Distributed Device Buses defined over the network. Agent is a software module that runs on every participating computer in the network, fulfilling its responsibilities, either as a Master node, or Provider node or both. As part of responsibilities of Provider node, Agent monitors activities of its hardware ports that are assigned to DDBs defined over the network and update the status of the ports in the State Manager accordingly. As part of responsibilities of the Master node, the Agent subscribes with State Manager for status changes of ports assigned to its DDBs and responds to those status changes.

The Agent is a light-weight software module, that uses *libudev* [21] Linux library for monitoring hardware events on usb ports and *hiredis* [22], a Redis client library implemented in C programming language to subscribe for event notifications from Redis. By updating State Manager with device related hardware events, subscribing to status changes events for relevant ports and responding to those status changes,

an Agent contributes to the functionality of Event Management Layer as explained in Section 7.2 and Service Matching Layer as explained in Section 7.3.

The following subsections describe how the combination of State Manager and Agent realize the functionality of Event Management Layer, Service Matching Layer and Distributed Device Bus Management Layer as explained in sections 7.2, 7.3 and 7.4 respectively.

8.2.3 Event Management Layer

An agent monitors device events on its hardware ports that are assigned to a Distributed Device Bus. Any device event noticed on the hardware port is updated in the State Manager. This status update in State Manager triggers notifications to the Master node. The Agent on the master node responds to this status update. Thus, a hardware event that originated in provider node is notified to master node via the state manager. Hence, agents and state manager communicate and coordinate to provide the functionality of event management layer.

8.2.4 Service Matching Layer

In our implementation, USBIP used at Device Virtualization Layer only supports remote device access but not remote port. To complement USBIP and support device plug-and-play on remote devices, service matching layer leverages the functionality of agent and state manger.

As explained earlier, the agent monitors device events on its hardware ports that are assigned to a distributed device bus. When a device is plugged in, agent updates the state manager about the availability of device. This status update on the port triggers a notification from state manager to master node regarding the availability of device. Similar communication happens when a device is plugged out. Hence, while USBIP is leveraged to support remote access for a device, functionality of USBIP is

complemented by agent and state manager to support remote ports.

8.2.5 Distributed Device Bus Management Layer

State manager contains the information of all the distributed device buses defined over the network along with ports assigned to each of the distributed device buses. Also, a tool called *dbus* is provided that provides a simple interface for users to create and manipulate distributed device buses in the network. This tool creates appropriate entries in state manager, assigning/deleting hardware ports to distributed device buses. These entries initiate required actions on agents.

Thus, state manager and the tool dbus provide the functionality of Distributed Device Bus Management Layer.

This event based notifications and distribution of responsibility to agents based on the relevant events has a five fold advantage:

1. Reduction in network traffic

   Agents update state manager with relevant hardware events on ports and also subscribe for relevant events from state manager. State manager is accessible from every node in the network and to dispatch notifications to appropriate agents. Agent's reliance on state manager eliminates the need for an agent to individually communicate with each provider/master node in the network for relevant updates. Hence, the communication in the network is reduced to bare minimum.

2. Data Consistency

   Since state manager stores all the required information of distributed device buses in the network and provides convenient interface to manipulate the data, agents do not locally store this information. Since no data is replicated across multiple agents, it is simpler to maintain the consistency of data.

3. Reduction in Overhead

   Agents on each node monitors device activity on ports assigned to distributed device buses and only subscribe for relevant events from state manager. Also, agents do not have to store any information about distributed device buses and do not have to individually communicate with each master/provider node in the network. Hence, the overhead of an agent on a node is directly proportional to its participation in the device consolidation framework and does not include any other overhead.

4. Compatibility with Existing Device Virtualization Solutions

   As demonstrated, our implementation of Distributed Device Bus infrastructure for USB devices leverages USBIP, an existing device virtualization solution. This approach, while providing the benefit of reusing the existing components, also ensures that any improvements in one layer can be easily incorporated without affecting the functionality in other layers.

5. Autonomous Agent

   Since agents on each node rely on state manager for notifications about relevant events, the functionality of an agent is independent of other agents in the network.

9. IMPLEMENTATION

For demonstrating the concept of *Distributed Device Bus*, we provided a working implementation that consolidates usb devices distributed across a network. This implementation serves as a proof of concept. The following sections describe various components developed and used as part of the implementation.

9.1 Agent

Agent is the software component that executes on each of the participating nodes. An agent has two important components:

1. Device Module

   Device module is responsible for observing device activity on hardware ports and notifying the agent accordingly. Our implementation used *libudev* [21] Linux library to monitor for usb device related events. Libudev provides simple interface to subscribe for device specific notifications.

   Device module enables to encapsulate device specific functionality. Hence, multiple device modules can be loaded to support different types of devices without affecting other components.

2. Datastore Module

   Data store module is responsible for subscribing to relevant event notifications from state manager and notify agent accordingly. Our implementation used Hiredis [22] a Redis client developed in C language. Hiredis provides a simple interface to subscribe and receive notifications from Redis.

   Data store module abstracts away the specific data store related logic from agent. Hence, any changes/updates to state manager can be managed by cor-

responding changes/updates to data store module on agents, without affecting the rest of the functionality of the agent.

9.2 State Manager

Redis version 3.0.1 [20] is used as a state manager. Redis is a open source key-value store and provides a simple yet powerful infrastructure that fulfills the requirements as specified in section 8.2.1. The key-value entries in data store have the form:

Key: [busname] : [master-node] : [protocol] : [provider-node] : [port-id]

Value: [status]

The meaning of the entries are explained in section 9.2.1. This simple key-value structure is chosen to leverage the notification mechanism in Redis. Redis provides primitive notification support. Notifications just identify the key that has been modified. So the above mentioned key structure is chosen, which makes it easy to identify the responsible nodes (master node and provider node) that have to act on the status change.

9.2.1 Information in Data Store

This section specifies the kind of information that should be available in the data store. Information about a distributed device should contain the following details:

1. Name

   Name of the DDB, it could be any identifier with alpha numeric characters. Name is just to recognize the bus.

2. Master Node

   Master node identifies the node that the DDB is assigned to.

3. Protocol

   Protocol specifies the kind of device and device virtualization used. In the present case, usb devices are used and usbip is used for device virtualization.

4. Provider Node

   Address of the provider node that contributes the device port to the DDB.

5. Port id

   Port id is the identifier of the port on the provider node.

6. Status

   Status represents the status of port with respect to distributed device bus. Value of status invokes specific actions from either of the master node or provider node. Status can have the following values:

   (a) ADDED

   This is the initial state of ports assigned to a distributed device bus. At the time a port is assigned to a distributed device bus, there may not be any device attached to port. As long as no device is plugged into the hardware port on the provider node, the status of the port remains ADDED.

   (b) EXPORTED

   When a device is plugged into the hardware port or if the device is already present, the provider node loads the stub driver for the device (as discussed in section 8.1) instead of device driver. At this point, device is ready to be accessed by the master node and agent on the provider node updates the status to EXPORTED.

(c) IMPORTED

A status update of EXPORTED triggers a notification to master node. Master node responds by leveraging the usbip infrastructure to access the remote device. If it succeeds, the status of port is updated to IMPORTED, implying that master node is now accessing the device on remote port. Otherwise, a status is updated to DISCONNECTED.

(d) DISCONNECTED

Any error while the master node accesses the remote device leads to status of the port being updated to DISCONNECTED. Errors could be problem with network access or abrupt removal of device or any error with usbip that unloads usbip drivers from master and provider nodes.

(e) TO_DELETE

Distributed Device Bus supports dynamic addition/deletion of hardware ports. A status of TO_DELETE indicates that user intends to delete the port from the distributed device bus. When the status of the port is modified to TO_DELETE, the master node responds by deleting the virtual device and notifies the applications that the device is unaccessible. Similarly, the provider responds by unloading the stub driver for the device and loads the actual device driver for the device. Once the stub driver for the device is unloaded, agent on provider node updates the status of the port to DELETED.

(f) DELETED

A status of DELETED signifies that the port is no longer assigned to distributed device bus and hence master node can no longer access the device on remote port and the device is only accessible locally on provider

node.

Figure 9.1 illustrates the state diagram of status of port in a distributed device bus as reflected in data store.

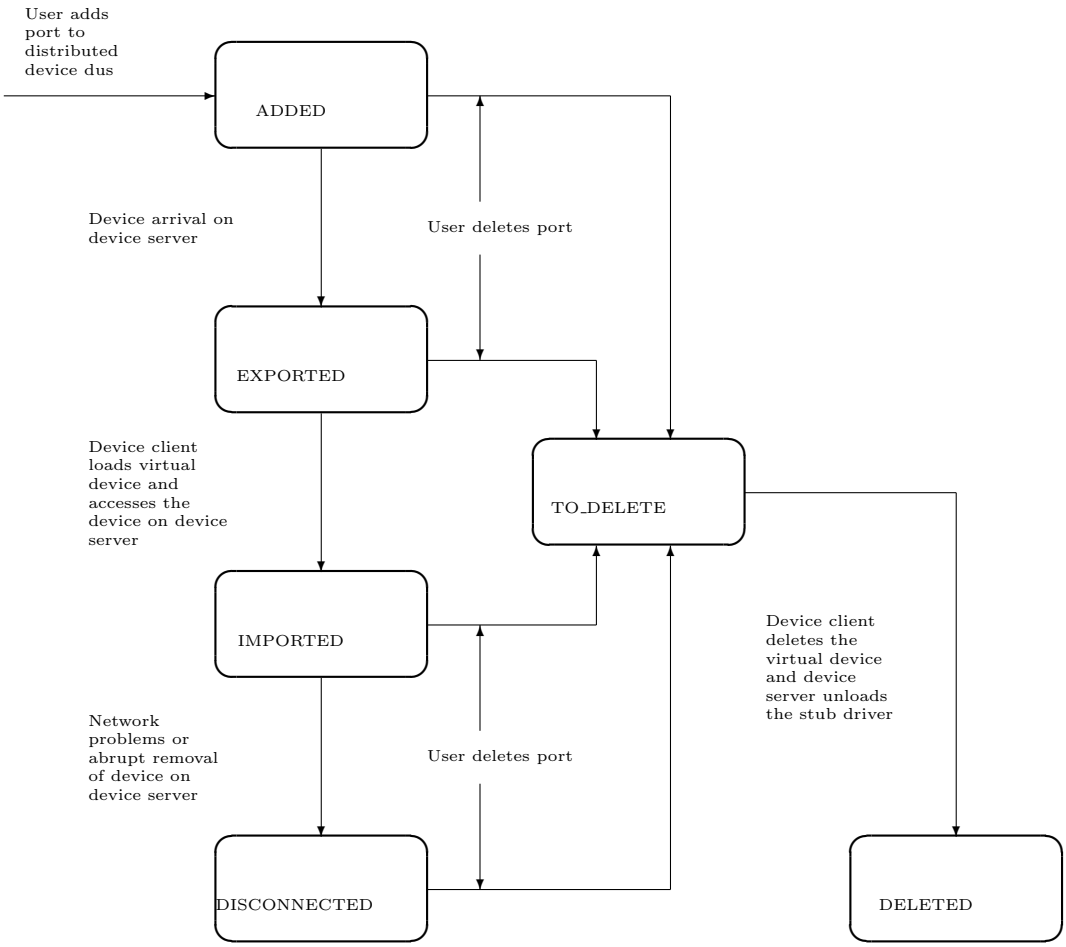

Figure 9.1: Port Status State Diagram

10. EVALUATION

10.1 Results

A number of experiments have been setup to assess the level to which solution criteria as discussed in section 2 has been satisfied. This implementation of Distributed Device Bus for usb devices is tested with a USB flash drive and USB web cam. A description of our evaluation for each of the criteria is provided below.

We faced some problems with usbip while using usb flash drives. Usbip support for flash drives has been inconsistent and only supported very few kinds of flash drives. These problems are specific to usbip and existed even outside our implementation environment. However, we could confirm that our implementation behaved reliably and updated the status of the port to DISCONNECTED when usbip was not able to support the usb flash drives.

10.1.1 Device Agnostic Service

Device agnosticism is demonstrated by using different type of USB devices. We used both a USB web cam and USB flash drive. USB web cam and USB flash drive attached to different computers are dynamically added to distributed device bus and applications running on master node successfully accessed these devices. We used Cheese [23] application for web cam and Ubuntu window manager for flash drive. Applications were not aware if they are accessing local devices or remote devices. Also the implementation provides the guide lines to extend the support for other devices.

10.1.2 Transparent and Complete Functionality

The applications accessing the remote devices were not aware that they are accessing remote devices and were able to support all the features of the device. Also, the device drivers loaded for the remote device on master node is the same as the device driver loaded for a local device. We used both USB web cam and USB flash storage device to demonstrate the complete functionality of remote devices.

10.1.3 Operating System Agnostic Service

We were unable to demonstrate the Operating System Agnosticism of the present implementation of distributed device bus for usb devices. However, this is not a limitation of the architecture of distributed device bus, but a limitation of usbip, the usb device virtualization solution and Redis, the state manager. Although USBIP provides limited support for windows, a computer running windows operating system can only act as device client. Besides, there is no official support for Redis on windows and even the unofficial version of Redis for windows doesn't have any client API required for the development of Agent. Hence a working demonstration of Operating System agnosticism was not possible.

10.1.4 Plug and Play

Plug and play events for USB devices are communicated across the network. Device ports dynamically allocated to distributed device bus are able to support plug and play events of devices on remote computers. The plug and play functionality is demonstrated with both USB flash and USB web cam devices.

10.1.5 Dynamic Reconfiguration

The implementation of Distributed Device Bus for usb devices supports dynamic reconfiguration of ports. Device ports can be dynamically added or deleted from a

distributed device bus without disturbing the other devices on the distributed device bus. This functionality is demonstrated by both USB web cam devices and USB flash storage devices.

11. SUMMARY

We proposed Distributed Device Bus, an architecture for a device consolidation framework that provides a simple and effective tool for managing devices distributed across a network. In Distributed Device Bus, access to a device is no longer limited to a single computer that has the device physically plugged in. Instead, the device can be configured to be transparently accessible from any computer in the network. This architecture is independent of device type and operating system, along with supporting transparent and complete functionality of the device. Distributed Device Bus naturally extends the plug-and-play functionality of devices over the network.

Our implementation of Distributed Device Bus for USB devices serves as a proof of concept along with demonstrating how existing device virtualization solutions can be leveraged to provide the required functionality. Our implementation supports transparent and complete functionality of all kinds of USB devices and extends the USB device plug-and-play capabilities over the network, allowing for dynamic reconfiguration of Distributed Device Buses defined over the network.

12. FUTURE WORK

Our implementation of Distributed Device Bus can be extended to support other device types like SCSI, PCI etc. Device virtualization solutions can be extended to support better communication mediums like Ethernet and Infiniband. This enables device virtualization solutions to leverage the low latency environments of Ethernet, Infiniband and provide better user experience while accessing remote devices. In the present implementation, VHCI driver on a participating computer acts independent of the Distributed Device Bus. The present implementation can be extended to improve transparency by connecting the VHCI driver to the corresponding Distributed Device Bus.

REFERENCES

[1] Wikipedia, "Peripheral Bus." http://en.wikipedia.org/wiki/Peripheral_bus, Accessed: June 2015.

[2] Wikipedia, "Wireless Sensor Network." http://en.wikipedia.org/wiki/Wireless_sensor_network, Accessed: June 2015.

[3] Cory Janssen, "Universal Serial Bus (USB)." http://www.techopedia.com/definition/2320/universal-serial-bus-usb, Accessed: June 2015.

[4] PCI Special Interest Group, "PCI Local Bus Specification, Revision 2.2." http://www.ics.uci.edu/~harris/ics216/pci/PCI_22.pdf, December 1998.

[5] R. O. Weber, "SCSI Architecture Model-3." http://ftp.t10.org/ftp/t10/document.02/02-119r0.pdf, March 2002.

[6] D.Daniel and J.Hui, "Virtualization of Local Computer Bus Architectures Over the Internet," in *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pp. 1884-1889, November 2007.

[7] Modbus Organization, "Modbus Application Protocol Specification." http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf, April 2012.

[8] Modbus Organization, "Modbus Messaging on TCP/IP Implementation Guide." http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf, October 2006.

[9] Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara, "USB/IP - A Peripheral Bus Extension for Device Sharing over IP Network," in *USENIX Annual Technical Conference, FREENIX Track*, pp 47-60, April 2005.

[10] Microsoft, "USB Request Blocks (URB)." https://msdn.microsoft.com/en-us/library/windows/hardware/ff537056(v=vs.85).aspx, Accessed: June 2015.

[11] Rakesh Roshan, "Remote USB Ports." http://oaktrust.library.tamu.edu/bitstream/handle/1969.1/151722/ROSHAN-THESIS-2013.pdf, December 2013.

[12] Mosaic Technology, "iSCSI Guide." http://www.mosaictec.com/pdf-docs/whitepapers/iSCSI_Guide.pdf, February 2009.

[13] Wikipedia, "Software Bus." http://en.wikipedia.org/wiki/Software_bus, Accessed: June 2015.

[14] Object Management Group, "Information Technology - Object Management Group Common Object Request Broker Architecture (CORBA) Interfaces," http://www.omg.org/cgi-bin/doc?formal/2012-05-03.pdf, May 2012.

[15] Stephane Chatty, "The Ivy software bus." http://www.eei.cena.fr/products/ivy/documentation/ivy.pdf, Accessed: June 2015

[16] Institutt for Energiteknikk, OECD Halden Reactor Project, "The Software Bus." http://www2.hrp.no/swbus/documentation/usersguide.pdf, Accessed: June 2015

[17] Dr. Silvano Maffeis, "Components Need Software Bus Middleware." https://proj-pssl.web.cern.ch/proj-pssl/projects/middleware/training/ibus.ppt, Accessed: June 2015

[18] J.A. Bergstra, P.Klint, "The Discrete Time TOOLBUS - A Software Coordination Architecture," in *Science of Computer Programming*, Volume 31, pp 205-229, July 1998.

[19] Wikipedia, "Enterprise Service Bus." http://en.wikipedia.org/wiki/Enterprise_service_bus, Accessed: June 2015.

[20] Salvatore Sanfilippo, "REDIS." http://www.redis.io/, Accessed: June 2015.

[21] The Linux Kernel Organization, "LIBUDEV." https://www.kernel.org/pub/linux/utils/kernel/hotplug/libudev/ch01.html, Accessed: June 2015.

[22] Salvatore Sanfilippo, Pieter Noordhuis, Matt Stancliff, Jan-Erik Rediger, "HIREDIS." https://github.com/redis/hiredis/, Accessed: June 2015.

[23] Canonical, "CHEESE." https://apps.ubuntu.com/cat/applications/precise/cheese/, Accessed: June 2015.

APPENDIX A

APPLICATION PROGRAMMING INTERFACE

The chapter aims to explain the Application Programming Interface available to users to manipulate DDBs. Section A.1 explains the tool *dbus* and its options. Section A.2 demonstrates the tool.

A.1 Manual

Users of the DDB create/delete a Distributed Device Bus and add/delete ports to Distributed Device Bus. A simple tool called *dbus* is provided for users. The options available for users of *dbus* are

dbus: a simple tool to manipulate Distributed Device Bus.

1. *-s –server* Specifies the address of the Redis data store. Defaults to 127.0.0.1
2. *-p –port* Specifies the port of Redis data store. Defaults to 6379.
3. *-b –bus* Specifies the name of the bus
4. *-m –master* Specifies the master node
5. *-c –command* Specifies the command to execute. The options are
   (a) *add:* adds a port to the specified bus on specified master node. Creates a new bus on adding first port.
   (b) *delete:* deletes a port from the specified bus. Deletes the bus on deletion of the last port
6. *-p –provider* Specifies the provider node.
7. *-d –devid* Specifies the device identifier on the provider node.

A.2 Usage

This section explains a simple scenario of using the tool. The network is as follows

Node 1 : 128.194.131.66

Node 2 : 128.194.131.52

Redis : 10.201.133.212

1. Create a DDB with master node as *128.194.131.52* and assign port *1-3* on *128.194.131.66* to it.

   dbus -s 10.201.133.212 -p 6379 -b dbus1 -c add -m 128.194.131.52 -p 128.194.131.66 -d 1-3

   The above command results in creation of the following entry in Redis:

   dbus1:128.194.131.52:usbip:128.194.131.66:1-3 ADDED

2. Plug in a usb device (say web cam) into port *1-3* on node *128.194.131.66*

   This action results in two intermediate steps.

   (a) When Agent on node *128.194.131.66* notices device arrival on port *1-3*, stub driver will be loaded for the device and updates the Redis status to EXPORTED, resulting in the following entry in Redis.

   dbus1:128.194.131.52:usbip:128.194.131.66:1-3 EXPORTED

   (b) When Agent on node *128.194.131.52* notices the status changed to EXPORTED, a virtual device will be loaded and makes the device on port *1-3* on node *128.194.131.66* accessible on node *128.194.131.52* and updates the status on Redis to IMPORTED, resulting in the following entry in Redis.

   dbus1:128.194.131.52:usbip:128.194.131.66:1-3 IMPORTED

3. At this point the device is accessible to any applications on node *128.194.131.52*

4. Delete the port assigned to DDB using the command, *dbus -s 10.201.133.212 -p 6379 -b dbus1 -c delete -m 128.194.131.52 -p 128.194.131.66 -d 1-3*

   This command results in updating the status in Redis to TO_DELETE

   dbus1:128.194.131.52:usbip:128.194.131.66:1-3 TO_DELETE

   This action results in two intermediate steps.

   (a) When agent on node *128.194.131.52* notices the status change to TO_DELETE, it deletes the virtual device and applications can no longer access the device.

   (b) When agent on node *128.194.131.66* notices the status change to TO_DELETE, it unloads the stub driver for the device and updates the status in Redis to DELETED resulting in the following entry in Redis.

   dbus1:128.194.131.52:usbip:128.194.131.66:1-3 DELETED