v9fb: A remote framebuffer infrastructure for Linux

Abhishek Kulkarni, Latchesar Ionkov
Los Alamos National Laboratory[*]
{kulkarni,lionkov}@lanl.gov

ABSTRACT

v9fb is a software infrastructure that allows extending framebuffer devices in Linux over the network by providing an abstraction to them in the form of a filesystem hierarchy. Framebuffer based graphic devices export a synthetic filesystem which offers a simple and easy-to-use interface for performing common framebuffer operations. Remote framebuffer devices could be accessed over the network using the 9P protocol support in Linux. We describe the infrastructure in detail and review some of the benefits it offers similar to Plan 9 distributed systems. We discuss the applications of this infrastructure to remotely display and run interactive applications on a terminal while offloading the computation to remote servers, and more importantly the flexibility it offers in driving tiled-display walls by aggregating graphic devices in the network.

1. Motivation

The framebuffer device in Linux offers an abstraction for the graphics hardware so that the applications using them do not have to bother about the low-level hardware interface to the device. Since the framebuffer is represented as a character device, a userspace application can open, read and write to it as a regular file. However, performing several routine graphic device operations like setting the resolution, fetching the color palette involves making use of a device-specific *ioctl* system call. This makes it difficult to export these devices as a network filesystem hierarchy.

Several remote display protocols for exchanging graphics over the network already exist. The widely used X window system in Linux is inherently based on a client-server model and implements the X display protocol to exchange bitmap display content between the client and the server. It, however, has been a target of much criticism since the early days[2] because of its overly complex architecture, lack of authentication in the protocol and the limited configurability in its client-server setup. Exporting raw pixel data of the framebuffer device makes it possible to run a window system on the CPU server. With the recent ongoing work on per-container device namespaces in the Linux kernel, this infrastructure provides the foundation for implementing a multiplexing window system similar to Rio [8] for Linux.

Remote display provides a way to interact with geographically distributed resources which are not within the physical proximity of the user. In addition to being used for remote display, v9fb can also be used in a few other interesting scenarios where it is not possible to use these other protocols. For instance, v9fb provides an alternative to monitoring the boot process of a remote machine in a network. This helps in cluster environments where the nodes are not equipped with a serial console to check the boot activity remotely. The booting node mounts the remote framebuffer device exported by the control node and the console of the node is mapped onto the remote framebuffer.

The main motivation for this infrastructure is to ease the setup of tiled-display walls for modeling and simulation of scientific data. High-resolution displays are increasingly being used for visualization of large datasets stored at a central storage facility. Display walls made out of commodity clusters are closely tied to the display nodes and do not allow for dynamic configurations. Developing simulation and modeling applications for these high-resolution tiled display walls is typically done using message passing libraries, new programming models or

[*]LANL publication: LA-UR-08-05604

software that use proxies to stream graphic commands over the network [12]. v9fb transparently aggregates the graphic devices in a network and exports a network attached framebuffer thus allowing greater flexibility in setting up a visualization cluster. Network-centric visualization is invariably favored since it ensures integrity and security of the data being maintained at a central location [7]. The application program is provided with a single logical view of the framebuffer device and thus requires no modifications to its code.

2. Introduction

Everything in Plan 9, including the graphics infrastructure, is implemented as a file server [9]. The file metaphor describes a well-defined interface to interact with all the resources in a distributed system. This makes it easy to work with the system, keeping it simple yet powerful. Raster graphics capability in Plan 9 is provided by devices like /dev/draw, /dev/screen and /dev/window. Along with the input and console devices, Plan 9 offers a highly configurable and customizable window system that works equally well over the network [8].

Despite considerable efforts, graphics in Linux remains poorly integrated with the rest of the system. The limitations of running the X server as a super user (root) further allows security loopholes which could be used to compromise the system. The framebuffer device abstraction was introduced in Linux starting with kernel version 2.1.107 [13]. The framebuffer device is an abstraction for the graphics hardware and is responsible for initializing the hardware, determining the hardware configuration and capabilities, allocating memory for the graphics hardware and providing common routines to interact with the graphics hardware. The Linux kernel contains drivers that support several different video hardware devices. The v9fb infrastructure exports the raw framebuffer memory and its operations as files. This model could be further extended to support specialized graphics hardware like Graphics Processing Units (GPUs).

The Linux kernel 2.6 offers support for the 9P protocol in the form of loadable kernel modules [1]. This allows the kernel to communicate with synthetic fileservers using the 9P distributed resource sharing protocol. v9fb leverages this support to implement a pseudo-framebuffer device which acts as an in-kernel 9P client that communicates with a framebuffer fileserver. The framebuffer appears as a regular character device to the applications using it. Every operation on this device is transparently translated into a 9P message that is sent across to the remote framebuffer fileserver. v9fb can work on any of the transport mechanisms like TCP or virtio offered by the 9P2000 implementation in the Linux kernel.

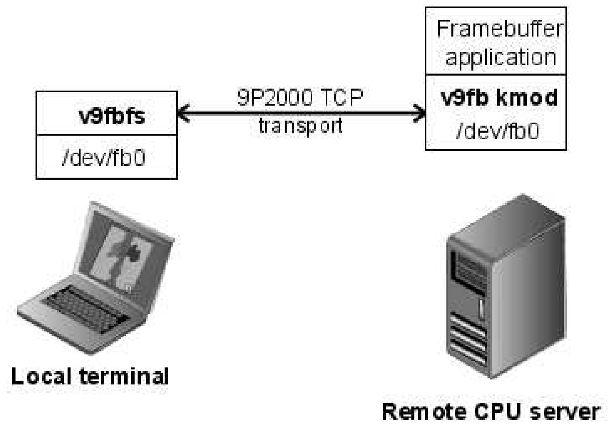

Figure 1: The local framebuffer device is exported by v9fbfs and mounted in the namespace of a remote CPU server which can draw to the remote device

The synthetic framebuffer filesystem *v9fbfs* exports a hierarchy that corresponds to various framebuffer operations which can be executed just by reading off or writing to these files. This also allows the framebuffer devices to be mounted locally and to interact with them as if they were local devices as shown in Figure 1. *v9fbfs* runs on all the display nodes in a visualization cluster and permits a highly-configurable and dynamic setup in which remote display devices can be attached or detached to rendering nodes based on their processing load. v9fb is scalable and can be optimized to support many display devices driving a tiled display wall with an effective resolution of over million pixels.

Coupled with the XCPU cluster management framework [5], this provides a holistic high-performance visualization environment that is easy to monitor and maintain. It allows a clear segregation of the display nodes from the render nodes and supports heterogeneous display hardware setup as a result of the framebuffer abstraction.

In many cases, simple pixel-based remote display can deliver superior performance than the more complex designs [15] based on other thin-client platform designs. The framebuffer synthetic filesystems allow adding multiple layers above the framebuffer much easier. Compression, encryption or the support for high-level drawing primitives on top of the framebuffer can be easily added without affecting the whole model.

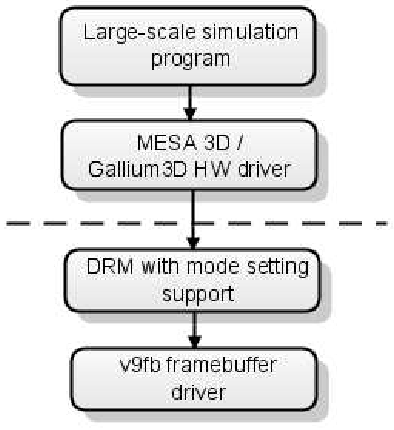
Large-scale simulation program
MESA 3D / Gallium3D HW driver
DRM with mode setting support
v9fb framebuffer driver

Figure 2: Running simulation and modeling programs directly on a hardware-accelerated framebuffer in absence of the X11 window system

Hardware-accelerated framebuffer makes use of the GPU operations to render graphics on the framebuffer device. Several libraries can use the framebuffer as a target to display high-resolution 2D and 3D graphics. With some of the upcoming changes in the Linux graphics stack like the changes in DRM (Direct Rendering Manager) and Gallium3D, the new proposed architecture for 3D graphics drivers, it would be much easier to display 3D hardware-accelerated graphics on the framebuffer without needing the X server at all as shown in Figure 2. As The framebuffer can be utilized as a drawing surface by the OpenGL applications, the X server and many other graphic drawing libraries like Simple DirectMedia Layer (SDL) or General Graphics Interface (GGI).

The remainder of this paper is organized as follows. In Section 3, we look at some of the related work on remote visualization systems and network-attached framebuffers. Section 4 offers a detailed design overview of the v9fb infrastructure describing how each component in the system interacts with the others. The actual implementation details are discussed in Section 5. We conclude by mentioning some of the future work in the last section.

3. Related Work

A number of existing proprietary solutions for remote visualization are available. Along with parallel graphics rendering toolkits and cluster management tools, these solutions provide a complete software environment for large-scale modeling and simulations. HP's Remote Graphics software, Sun's Visualization System and SGI's Remote Visualization are among many other proprietary solutions that offer remote access to 2D and 3D graphics. Most of these remote display solutions primarily rely on VNC which uses the Remote Framebuffer Protocol (RFB) to exchange display updates over the network.

Tiled display walls usually use pixel-based streaming software to stream the rendered data to the display nodes or a network attached framebuffer. The Scalable Adaptive Graphics Environment (SAGE), developed at the University of Illinois Chicago, is a distributed visualization architecture specifically designed for decoupling graphics rendering from the graphics display [6]. SAGE dispatches visualization jobs for rendering to the appropriate resource in a cluster and streams the resultant pixel data to the remote display. Others, like TeraVision, JuxtaView also provide an infrastructure for remotely displaying imagery in a cluster.

OpenGL toolkits for cluster-based rendering like Chromium [3] or VirtualGL use techniques like function call interposing to "snoop" the OpenGL protocol and transfer it over the wire to the remote proxies in a cluster. This techniques make it difficult to keep up with the evolving standards and specifications described by OpenGL and add to the overhead in terms of complexity of the architecture.

IBM's Scalable Graphics Engine (SGE-3) offers a hardware-based approach to a network-attached framebuffer[10, 14]. It aggregates the pixel data generated by a rendering cluster to drive a high-resolution tiled display wall. Several other sort-first rendering systems like WireGL allow unmodified graphics application to be scaled to work on a high-resolution tiled-display.

4. Design Overview

The Linux framebuffer was designed to extend a hardware-independent abstraction to the underlying graphics hardware. Some hardware running Linux did not support the VGA text mode, and the framebuffer proved to be a device-independent way of emulating a text console on these machines. Besides, since the VGA fonts could only cover a 512 character-set simultaneously there was no way to represent UTF-8 on the Linux console. The framebuffer support brought in Unicode console terminal emulators to Linux and more importantly the ability to do graphics without having to rely on the overly loaded X Window system.

The Linux framebuffer device exports the graphic device's raw memory which can be directly accessed from a userspace application. Even though this device memory can be accessed using the basic file operations, Linux has not excessively made use of the file metaphor to provide a generic and consistent way of controlling devices. A framebuffer device defines several device-specific *ioctls* which could be used to initialize or control the device. A typical application to draw a pixel to the framebuffer device is shown below.

	Framebuffer Operation	**Programmatic equivalent**
1	Open the framebuffer device.	`fd=open("/dev/fb0",O_RDWR)`
2	Get fixed screen information	`ioctl(fd,FBIOGET_FSCREENINFO,&f)`
3	Get variable screen information	`ioctl(fbfd,FBIOGET_VSCREENINFO,&v)`
4	Determine screen resolution	`size=v.xres*v.yres*v.bits_per_pixel/8`
5	Map the framebuffer memory	`mmap(0,size,PROT_WRITE,MAP_SHARED,fd,0)`
6	Draw to the mapped memory area	`*(fbp+` location)=32
7	Close the framebuffer device	`close(fd)`

Table 1: Basic framebuffer operations and their programmatic equivalents for drawing a pixel on the framebuffer. This table is just to illustrate the flow of operations involved when accessing the framebuffer device.

The motivation of v9fb was to eliminate the specialized ad-hoc interface to the framebuffer device in favor of the more familiar file-centric device interface. The exported framebuffer device interface is textual rather than binary and can be mounted to a remote machine over the network. It provides specialized files that can be accessed to control the framebuffer device, fetch framebuffer device information or draw to the framebuffer.

In addition to providing a hierarchical framebuffer filesystem, v9fb also provides the infrastructure to run unmodified Linux framebuffer applications through the v9fb kernel framebuffer driver. This allows the existing framebuffer graphic applications like modeling and simulations programs to seamlessly utilize a distributed environment for rendering or displaying.

The v9fb infrastructure consists of several entities interacting with each other to make the process of accessing remote framebuffer devices as transparent as possible.

- v9fbfs
- v9fb kernel module
- v9fbaggr
- v9fbmuxfs

v9fbfs is a userspace 9P fileserver that exports a filesystem hierarchy of the framebuffer. The *v9fb kernel module* creates a virtual framebuffer device that acts a 9P client translating all the framebuffer operations into POSIX-like file I/O operations. These calls are forwarded to either to *v9fbfs* or *v9fbaggr* over the 9P protocol. *v9fbaggr* is another userspace 9P fileserver which aggregates the framebuffer resources provided by multiple *v9fbfs* fileservers to export a logical view of a single large framebuffer. *v9fbaggr* offers an exactly similar interface as *v9fbfs* thus making it seamless to communicate with the *v9fb kernel module*.

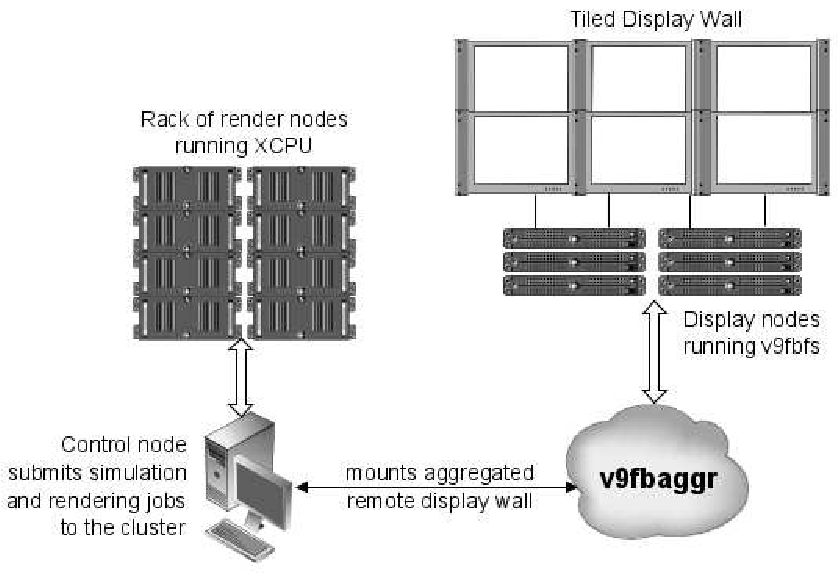
Tiled Display Wall
Rack of render nodes running XCPU
Display nodes running v9fbfs
Control node submits simulation and rendering jobs to the cluster
mounts aggregated remote display wall
v9fbaggr

Figure 3: High-performance computing environment for large-scale modeling and simulations using XCPU and V9FB

Figure 3 shows a typical setup of a rendering cluster environment using XCPU and V9FB. At the first glance, the control node appears as a potential bottleneck in this environment. However, the control node only acts as a front-end for submitting jobs. With support for dynamic namespaces offered by XCPU, the aggregated framebuffer device could be mounted in the namespace of each rendering node which directly writes on to a specific framebuffer of the display wall.

v9fbmuxfs is a userspace 9P fileserver which is almost similar to *v9fbfs*. *v9fbmuxfs* divides a framebuffer into multiple regions exporting each as a logical framebuffer device. It multiplexes the access to each of these regions to simultaneously display the framebuffer output from several clients. Since most modern graphic cards support tiled framebuffers, each tile could be rendered by different machine to achieve a much faster performance.

v9fb offers secure delivery of the display data since it uses the authentication support in 9P2000 protocol. The 9P auth information negotiates authentication between the client and the fileserver before exchange of raw pixel data takes place. The ordered delivery of messages in 9P protocol ensures there is no corruption of the frame pixels. Synchronization has not been taken into account but could easily be added into v9fb.

Synthetic fileservers allow easy addition and removal of functional layers to the architecture. These can further be in the form of fileservers or simple libraries acting on the exported files. For instance, to make efficient use of the network bandwidth the raw pixel data transferred over the network can be compressed before sending. Several performance optimization techniques have been taken into account to achieve a good performance.

4.1. Performance Optimization

v9fb has been designed with low-latency high-bandwidth links in mind where the remote display nodes are connected to the control nodes using a suitably high-speed network interconnect like Gigabit Ethernet. Transmitting raw pixel data over the wire consumes considerable bandwidth for real-time visual applications like video streams and interactive simulations.

4.2. Framebuffer compression

The raw framebuffer data can be compressed using various compression algorithms before transmitting it across the network. This reduces the load on the network, however adds to the overhead of post-processing the data before displaying it on the framebuffer. Compression helps in low-latency links where the network gets overloaded by large bursts of raw pixel data. Video hardware has already started supporting compression at the device level to reduce power consumption [11]. Compression is done on a per-line basis by using a simple compression algorithm like run-length encoding (RLE) or the LZ77 algorithm.

4.3. Framebuffer caching

Caching the framebuffer data at the client can improve the performance in case of non-interactive applications where most accesses involve reading from a static framebuffer. A write to the remotely mounted framebuffer invalidates the cache, and the changes have to be propagated back to the framebuffer fileserver. Introducing caching, however, adds to unmanaged complexity and the performance increases are seldom guaranteed[15].

4.4. Double Buffering

Double buffering at the client and server side can improve performance in most cases. The network-attached framebuffer acts as a back buffer used by the framebuffer fileserver. The scanout buffer acts as a front buffer which represents the memory of the video device. Flipping between the two buffers compensates the network delay to a certain extent and can allow a continuous stream of frames on the video display.

4.5. Multiplexed operations

Multiple clients writing to a single framebuffer pose a potential bottleneck in performance. Multiple reads and writes can be multiplexed at the server with separate threads performing the operations at once. This would significantly add to the performance of *v9fbaggr* which essentially communicates to multiple framebuffer fileservers *v9fbfs* simultaneously. When multiple Treads or Twrites are to be done in parallel, multiple threads are spawned by the server handling these request in parallel.

5. Implementation

5.1. v9fbfs

v9fbfs, along with the other user-space framebuffer fileservers has been implemented using *libspfs*, a library belonging to the NPFS project[4] which facilitates writing 9P2000 compliant userspace fileservers in Linux. v9fbfs is a userspace 9P fileserver which scans the local machine for existing framebuffer devices and exports an interface in the form of a file hierarchy given below.

```
/ctl
/data
/mmio
/fscreeninfo
/vscreeninfo
/cmap
```

```
/con2fbmap
/state
```

5.1.1. `ctl` file

The `ctl` file is used to control the framebuffer server and perform some several framebuffer display operations. It supports the following commands :

pandisplay The pandisplay command is used to pan or wrap the display when the X or Y offset of the display have changed.

blank *blankmode* Blank the framebuffer based on the supplied blank mode. This could be used to suspend or power down remote idle displays to save power.

reload Reload the framebuffer filesystem interface. This looks for newly added framebuffer devices and exports them.

5.1.2. `data` file

The `data` file represents the actual raw framebuffer memory buffer usually represented by the /dev/fb[0-7] device in Linux. Writing to this file writes directly to the framebuffer memory. Similarly, this file is read to fetch the current framebuffer contents.

5.1.3. `mmio` file

This file represents the memory-mapped IO memory of the framebuffer device. Userspace applications can program the MMIO registers by reading or writing to this file. This can be used to provide hardware acceleration to the framebuffer from the userspace.

5.1.4. `fscreeninfo` file

Reading from this file retrieves the fixed screen information of the framebuffer graphic device. The device-specific framebuffer information like device type, visual properties, acceleration support, the framebuffer memory length and addresses, the length of the scanline in bytes and the memory-mapped I/O addresses of the device is exported by this file. Fixed information cannot be changed, thus this file cannot be written to.

5.1.5. `vscreeninfo` file

Reading from this file fetches the virtual screen information of the framebuffer. This can be used to determine the display capabilities of the framebuffer, supported resolutions and color palettes, acceleration flags, bits-per-pixel and the margin and sync lengths among other information. Any of the virtual screen information can be changed by writing to this file.

5.1.6. `cmap` file

Get/Put the color palette information.

5.1.7. `con2fbmap` file

Used to map the console onto the framebuffer device and vice versa.

5.1.8. `state` file

State of the framebuffer device which is used by *v9fbaggr* to maintain synchronization between multiple displays.

Reading and/or writing to a particular file invokes a corresponding framebuffer device-specific operation which talks to the underlying framebuffer device. This provides an alternative to using the ioctl system call for device communication and consequently allows the device to be accessed over the network. This filesystem interface exported by *v9fbfs* can also be mounted as a filesystem using V9FS.

```
$ ./v9fbfs -d
Found framebuffer device /dev/fb0 ...
/dev/fb0 : VESA VGA
Framebuffer device memory from 0xfb000000 to 0xfb600000
Length: 6291456 bytes
Framebuffer MMIO from (nil) to (nil)
Length: 0 bytes
listening on port 8883
```

By mounting *v9fbfs* as a filesystem, framebuffer applications can use this interface to draw to the framebuffer device. With recent support for per-process namespaces in Linux, it allows each process to have an exclusive view of the framebuffer device.

```
$ mount -t 9p 192.168.10.1 /mnt/fb -o port=8883, uname=abhishek, debug=511
$ ls /mnt/fb/fb0/
cmap  con2fbmap  ctl  data  fscreeninfo  state  vscreeninfo
$ cat /mnt/fb/fb0/fscreeninfo
VESA VGA
4211081216 6291456
0 0
2
0 0 0
4096
0 0
```

v9fbfs can handle multiple framebuffer devices (upto 8). Applications drawing on the top of the framebuffer usually accept a command-line parameter to draw to a different framebuffer device. Alternatively, the global FRAMEBUFFER environment variable can be set to use a different framebuffer device.

5.2. v9fbaggr

v9fbaggr is a userspace 9P server and client typically running on a control node. On startup, v9fbaggr reads a configuration file *v9fbaggr.conf* which specifies the remote framebuffer devices that it needs to aggregate and their relative geometry to export a single logical framebuffer device.

A typical configuration file for a 3x3 tiled display wall is shown below.

```
tile1=192.168.10.40!8883, tile2=192.168.10.64!8883, tile3=192.168.10.67
tile4=192.168.10.41!8883, tile5=192.168.10.65!8883, tile6=192.168.10.68
tile7=192.168.10.42!8883, tile8=192.168.10.66!8883, tile9=192.168.10.69
```

Currently, each newline in the configuration file represents a new row in the geometry of the tiled display wall. Each entry is represented by a nodename followed by its network address and the port on which the server is listening. Use of a rigid data representation format like s-expressions might be considered in the future.

v9fbaggr communicates to the framebuffer fileserver *v9fbfs* running on these machines, fetches their fixed and variable display information and aggregates the remote display resources to provide a logical view of the 3x3 tiled display wall as a single unit of display. Since, v9fbaggr exports an exactly similar interface as that of v9fbfs, application remain transparent of the underlying multiple display devices spread across the network. Framebuffer operations like panning the display, turning the display blank, reloading the fileservers are translated such that they apply to all the remote framebuffer devices aggregated by *v9fbaggr*. In addition to this, the commands accepted by the `ctl` file also takes an additional parameter, the node name, to which the operation is to be applied.

v9fbaggr implements a memory management unit to translate the virtual address of the aggregated framebuffer to an address of a specific framebuffer device based on the geometry and layout of the tiled display wall. The virtual aggregated framebuffer provides a contiguous linear memory to the application using it. Each memory access to this framebuffer is translated to a 9P read or write to the appropriate framebuffer fileserver. The framebuffer memory of remote framebuffer devices are represented as segments and mapped onto the virtual aggregated framebuffer exported by *v9fbaggr*. Memory accesses to this framebuffer pass through a segment selector which points to the various segment pointers depending on the actual layout of the framebuffer devices. *v9fbaggr* allows unmodified applications and programs to be run on a tiled display wall.

5.3. v9fb kernel module

The v9fb kernel module typically runs on the control node or the head node and creates a pseudo-framebuffer device which translates framebuffer device operations into corresponding 9P calls. The intended use of this kernel module is to mount the filesystem exported by *v9fbaggr* so that it can act as a passthrough framebuffer device to draw transparently to the tiled display wall. It could also be used to mount a single remote framebuffer device for remote workstation display applications.

```
$ modprobe v9fb address=192.168.1.40
$ dmesg | tail -n 2
[118398.958865] v9fb: Enabling remote framebuffer support
[118398.960945] fb1: Remote frame buffer device

$ rmmod v9fb
$ dmesg | tail -n 1
[118401.461253] v9fb: Unmounting remote framebuffer device
```

The kernel module has been written so that v9fb can support existing framebuffer applications without having to change them. It translates the device specific ioctl calls into a corresponding 9P call. For instance, to get the virtual screen information of a framebuffer device, the ioctl call to be used is as follows -

```
ioctl(fd, FBIOGET_VSCREENINFO, vscr);
/* vscr is a structure to hold the variable screen
information */
```

The v9fb kernel module translates this into an appropriate 9P operation to read from the `vscreeninfo` file as shown below.

```
<<< (0x8059660) Twalk tag 0 fid 3 newfid 4 nwname 1 'vscreeninfo'
>>> (0x8059660) Rwalk tag 0 nwqid 1 (0000000000000005 0 '')

<<< (0x8059660) Twalk tag 0 fid 4 newfid 5 nwname 0
>>> (0x8059660) Rwalk tag 0 nwqid 0
<<< (0x8059660) Topen tag 0 fid 5 mode 0
>>> (0x8059660) Ropen tag 0 (0000000000000005 0 '') iounit 0

<<< (0x8059660) Tread tag 0 fid 5 offset 0 count 8168
>>> (0x8059660) Rread tag 0 count 110 data 31303234 20373638 20313032
34203736 38203020 300a3332 20300a31 36203820 30203820 38203020 30203820
```

```
30203234 20382030 0a300a30 0a343239 34393637

<<< (0x8059660) Tclunk tag 0 fid 5
>>> (0x8059660) Rclunk tag 0
<<< (0x8059660) Tclunk tag 0 fid 4
>>> (0x8059660) Rclunk tag 0
```

This provides a way to serialize and deserialize device-specific framebuffer calls and obtain the equivalent functionality by marshalling these calls using 9P. Most of the framebuffer `ioctl()` calls are only done at the initialization time and once the display has b een setup properly, majority of the traffic involves reading from and writing to the framebuffer memory. Thus, multiplexing the reads and writes promises considerable performance gains.

5.4. v9fbmuxfs

v9fbmuxfs is similar to *v9fbfs* in a way that it exports the framebuffer device interface as a filesystem. It however divides a single framebuffer device into separate regions exporting each as a virtual framebuffer device which a client can write to. Simultaneous rendering and display of a single frame by multiple clients or multiple graphic processing units on a single client can be done with the help of *v9fbmuxfs*. A working implementation of *v9fbmuxfs* has been completed. It allows several framebuffer applications to simultaneously write to distinct regions of a single framebuffer under the assumption that the region is a framebuffer device itself. Several issues about applications contending to control the framebuffer device are yet to be addressed.

6. Future Work

Several issues still remain to be dealt with to use v9fb in a production visualization environment. Due to constraints in time, actual performance metrics for driving tiled display walls using v9fb could not be obtained by the time of this writing. Overall performance can be tuned using several ways discussed in Section 4. Apart from this, we are working to support the following features for the v9fb infrastructure.

6.1. Support for input events

Sending keyboard and mouse events over the network forms an integral part of remote display technologies. Currently, v9fb does not address the forwarding of input events over the network. Extending v9fb to support input events is trivial and we have started working on it.

6.2. Hardware-accelerated framebuffer

Due to the proprietary binary-only drivers distributed by major graphic card manufacturing firms like NVIDIA, it has become difficult to use hardware acceleration for the framebuffer. With several initiatives to revamp the state of graphics in Linux, it would soon be possible to use the framebuffer or the in-kernel Direct Rendering Manager (DRM) to draw to the video memory. DirectFB is a thin library which provides hardware graphics acceleration to the framebuffer. A DirectFB extension to v9fb would allow using hardware acceleration to draw high-resolution 3D graphics on the framebuffer device.

6.3. Communication between v9fbfs

One of the most common uses of the tiled display wall is to display high-resolution imagery. Moving and panning of images on the tiled display wall results in resending the pixel data from the control nodes to all the display nodes. This forms a potential bottleneck at the control node. Enabling communication between the individual framebuffer fileservers would increase the performance of interactive applications on the display wall.

7. Conclusion

v9fb provides a novel approach of accessing remote devices over the network in Linux using concepts and ideas employed by Plan 9 since its inception. Withstanding the several difficulties posed by the rigid device subsystem in Linux, this scheme could be easily extended to allow exporting various other devices as a filesystem over the network. v9fb finds various applications in high performance computing and remote visualization technologies. It offers flexibility and configurability leading to dynamic architectures in a large-scale modeling and simulation environment. We are working on several optimizations to this infrastructure to make it capable enough for use in production environments.

References

[1] Eric Van Hensbergen and Ron Minnich. Grave robbers from outer space: Using 9p2000 under linux. In *In Proceedings of Freenix Annual Conference*, pages 83–94, 2005.

[2] Don Hopkins. The X-Windows Disaster. *UNIX-HATERS Handbook*.

[3] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3):693–702, 2002.

[4] Latchesar Ionkov. The NPFS project. `http://sourceforge.net/projects/npfs`.

[5] Ronald Minnich and Andrey Mirtchovski. XCPU: a new, 9p-based, process management system for clusters and grids. In *CLUSTER*. IEEE, 2006.

[6] Krishnaprasad Naveen, Vishwanath Venkatram, Chandrasekhar Vaidya, Schwarz Nicholas, Spale Allan, Zhang Charles, Goldman Gideon, Leigh Jason, and Johnson Andrew. SAGE: the Scalable Adaptive Graphics Environment.

[7] Brian Paul, Sean Ahern, Wes Bethel, Eric Brugger, Rich Cook, Jamison Daniel, Ken Lewis, Jens Owen, and Dale Southard. Chromium Renderserver: Scalable and Open Remote Rendering Infrastructure. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):627–639, 2008.

[8] Rob Pike. Rio: Design of a concurrent window system. February 2000.

[9] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[10] Prabhat and Samuel G. Fulcomer. Experiences in driving a cave with IBM scalable graphics engine-3 (SGE-3) prototypes. In *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 231–234, New York, NY, USA, 2005. ACM.

[11] Hojun Shim, Naehyuck Chang, and Massoud Pedram. A compressed frame buffer to reduce display power consumption in mobile systems. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 818–823, Piscataway, NJ, USA, 2004. IEEE Press.

[12] Munjae Song. A Survey on Projector-based PC Cluster Distributed Large Screen Displays and Shader Technologies.

[13] Geert Uytterhoeven. The Linux Frame Buffer Device Subsystem. *Linux Expo '99*, 1999.

[14] Bin Wei, Douglas W. Clark, Edward W. Felten, Kai Li, and Gordon Stoll. Performance issues of a distributed frame buffer on a multicomputer. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–96, New York, NY, USA, 1998. ACM.

[15] S. Jae Yang, Jason Nieh, Matt Selsky, and Nikhil Tiwari. The Performance of Remote Display Mechanisms for Thin-Client Computing. In *In Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.