DisplayCast: a High Performance Screen Sharing System for Intranets

Surendar Chandra and Lawrence A. Rowe
FX Palo Alto Laboratory Inc.
3174 Porter Drive, Palo Alto, CA 94304
surendar@acm.org, rowe@fxpal.com

ABSTRACT

DisplayCast is a many to many screen sharing system that is targeted towards Intranet scenarios. The capture software runs on all computers whose screens need to be shared. It uses an application agnostic screen capture mechanism that creates a sequence of pixmap images of the screen updates. It transforms these pixmaps to vastly improve the lossless Zlib compression performance. These algorithms were developed after an extensive analysis of typical screen contents. DisplayCast shares the processor and network resources required for screen capture, compression and transmission with host applications whose output needs to be shared. It balances the need for high performance screen capture with reducing its resource interference with user applications. DisplayCast uses Zeroconf for naming and asynchronous location. It provides support for Cisco WiFi and Bluetooth based localization. It also includes a HTTP/REST based controller for remote session initiation and control. DisplayCast supports screen capture and playback in computers running Windows 7 and Mac OS X operating systems. Remote screens can be archived into a H.264 encoded movie on a Mac. They can also be played back in real time on Apple iPhones and iPads. The software is released under a New BSD license.

Categories and Subject Descriptors

J.m [**Computer Applications**]: Miscellaneous

General Terms

Experimentation

Keywords

Screencast, DisplayCast, Screen Capture, Screen Sharing

1. INTRODUCTION

DisplayCast targets collaboration scenarios in which many users seamlessly share their screen contents. It is motivated by the ubiquity of personal devices such as smartphones, tablets and desktops as well as large projectors that are embedded in the environment. High speed wireless LANs makes DisplayCast practical.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MM'12, October 29–November 2, 2012, Nara, Japan.
Copyright 2012 ACM 978-1-4503-1089-5/12/10 ...$15.00.

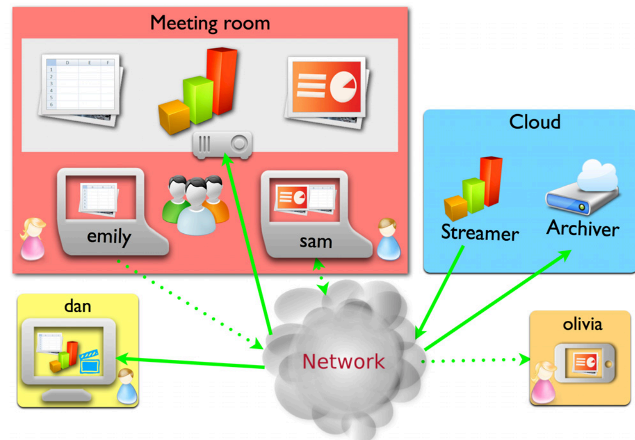


Figure 1: Usage scenario

1.1 Usage scenarios:

First, we describe two usage scenarios: one in industry and the other in academia to motivate the need for our system:

1.1.1 Group collaboration in a corporate setting

We illustrate our vision in Figure 1. Meeting rooms are replete with projectors and large monitors. In our own meeting room, we can project a 3840x1200 image using two projectors. Meetings frequently start with one presenter showing their ideas to the team. In Figure 1, *Emily* presents her *spreadsheet*. Others respond by projecting contents from their own personal devices (e.g., *Sam* projects powerpoint slides from his laptop). Physical proximity is not always possible; some team members monitor the conversation from their own offices. For example, *Dan* watches some of these presentations while simultaneously performing other tasks. Others (such as *Olivia*) may briefly leave the meeting to take a phone call (with her bluetooth headset) while continuing to watch the presentations on her smart phone. Some presentations originate in the cloud; all these presentations may also be archived to the cloud.

1.1.2 Lecture capture in academia

Powerpoint slides have become an integral component of lectures. In our vision, physical proximity is not mandatory; students join in remotely (say from the dormitory) to watch the lectures. Traditionally, students sought clarification using the in-class blackboards. In the future, they will illustrate and project them from their personal devices. Archiving all these interactions would also be invaluable for lecture review.

These usage scenarios highlight two important requirements for an Intranet focussed screen-sharing system:

- **flexibility:** device homogeneity requires a flexible screen sharing system. Approaches such as Apple AirPlay[2], Microsoft Media Redirection[**?**] and Access Grid Distributed PowerPoint [7] transfer the original objects (e.g., powerpoint documents, video files) to the remote users for rendering. They achieve good performance without additional capture and compression overhead. However, they require tight coordination between the sender's application and the remote renderer in order to enforce any DRM restrictions as well as faithfully decode the object. This places severe restrictions on the types of applications whose outputs could be shared. For example, Powerpoint users have to be certain that all their team members have the appropriate Powerpoint version along with all the embedded images, fonts etc. Hence, we use an approach that captures the screen contents as a sequence of images and then shares them. Remote viewing only requires tools that can play back these images.

- **high performance:** Since high speed networks are ubiquitous in Intranets, users want their screens to be faithfully reproduced to the remote users. VNC [6] shares pixmap updates using the remote framebuffer (RFB) distribution protocol [5]. RFB is optimized for constrained networks and only achieves delivery rates of 4.5 [8] in Intranets. Though users were willing to tolerate such rates for remote access scenarios, they found them inadequate for Intranet scenarios. Note that hardware limitations can also place insurmountable restrictions. For example, the Apple iPhone smartphones use a 65 Mbps single stream IEEE 802.11n wireless LAN even though laptops can connect to the same network (using sophisticated hardware) at link speeds of 600 Mbps.

1.2 Technical approach

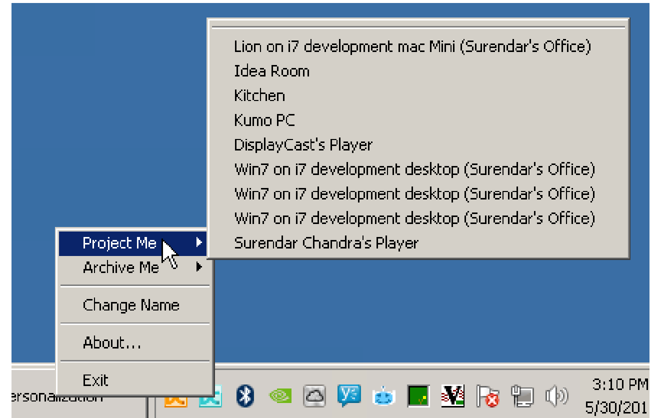
Initially, we used VNC [6] for capturing the screen contents. Our experience with a variety of deployed VNC servers confirmed the reported poor performance of VNC [8]. Inexplicably, the poor performance was deemed acceptable for some usage scenarios.

In order to understand the requirements for high performance screen capture, we investigated the nature of screen contents for various usage scenarios. We dual-booted two laptops to Mac OS X and Windows 7 in order to investigate the differences imposed by hardware capabilities as well as operating systems.

We showed [3] that screen updates exhibited long durations of inactivity. During inactive durations, VNC's poor performance [8] was acceptable. When active, screens were updated at far higher rates than was supported by VNC. The mismatch was pronounced for interactive scenarios. However, even when the screens were updated frequently, the number of updated pixels were small; thus allowing our system to capture all the screen changes. We showed that crucial information can be lost if individual updates were merged. When the available system resources could not support high capture rates, we showed ways to *effectively* coalesce pending updates.

We showed the inability of hardware assisted lossy H.264 encoding to capture the screen at high capture rates. We showed that Zlib lossless compression achieved poor compression performance for screen updates. High capture rates necessitated the need for a better approach. By analyzing the screen pixels, we developed a practical transformation that significantly improved compression rates. Our system captured 240 updates per second while only using 4.6 Mbps for interactive scenarios. Still, our approach could not

(a) OS X (streamers listed in blue are nearby; using BlueTooth)



(b) Windows 7

Figure 2: Streamer system tray user interface

achieve higher capture rates than was achieved by VNC while playing movies in fullscreen mode; the CPU remains the bottleneck.

The open sourced system incorporates the results of this analysis [3]. DisplayCast is deployed within the lab.

2. SYSTEM DESCRIPTION

DisplayCast system consists of two major components: screen capture and distribution as well as a naming and location management component. Notably absent are: a) an authentication and authorization framework; any DisplayCast user can share their screens and b) interaction mechanisms (such as keyboard and mouse events) that allow remote users to interact with applications; remote users can only watch the shared screen.

All applications are written in native code (C# for Windows and Objective-C for Mac/iOS) and do not share any code artifacts between the platforms. Our initial prototype used IP Multicast. However, creating a robust system required considerable fine tuning of our Cisco based Intranet networking fabric. Hence we switched to using TCP. Using a point-to-point TCP link for each source-receiver pairing limits the scalability of the current system.

2.1 Screen capture and Distribution

We provide three applications for screen capture and distribution: *Streamer* captures the screen and transmits them using TCP for real-time viewing by *Player*s. *Archiver* creates an H.264 movie of the *Streamer*. *Streamer* is available in Windows 7 and Mac OS X (10.6+), *Player*s are available for Windows 7, Mac OS X (10.6+) and iOS (5.0+). *Archiver* requires Max OS X 10.7+. Next we describe each of these programs in further detail.

2.1.1 Streamer

In Windows 7, the *Streamer* captures the screen update boundaries using the DemoForge mirror driver [1]. This driver implements the Windows mirroring function [**?**]. The driver intercepts and stores the boundaries of the 20,000 most recent updates. Based

on our analysis [3], the *Streamer* polls the driver every 16 msec to collect the stored updates. *Streamer* memory maps the mirror driver's framebuffer copy to access the pixels that correspond to each screen update. On Mac OS X, we used the *CGRegisterScreenRefreshCallback()* call to receive asynchronous notifications regarding screen updates, and *CGDisplayCreateImageForRect()* call to retrieve the pixel data.

We compress the captured pixels for transmission. We achieve higher compression efficiency by transforming the inter-update pixel temporal redundancy into intra-update spatial redundancy. Our analysis [3] showed that the following transformation improved Zlib [4] compression performance without incurring significant processing overhead. We separated screen updates into a similarity byte-map and pixel data. We use 0x00 and 0xFF to represent similar and dissimilar pixels in the byte-map, respectively. The pixel data is a sparse array that stores the 24 bit RGB values of dissimilar pixels. The resulting data is then compressed using Zlib. In Windows 7, we access Zlib using the .NET 4.0 - *DeflateStream()*. Our transformation improved the compression factor of 5.1:1 achieved by Zlib to 143:1; a 28 fold improvement in compression efficiency.

Streamers provide a snapshot of the current screen contents as a PNG image through a HTTP service; it is expected that the clients will use this snapshot service sparingly. Users can also achieve some privacy by specifying a update masking region; only screen updates within this region are streamed. This masking region can be specified through a service request to the *Streamer*.

Users interact with the *Streamer* using a task bar system tray interface. Task bar applications appear on the top right tray in Mac OS X (Figure 2(a)) and bottom right in Windows (Figure 2(b)). Users can use the "Project Me" command to send the local screen to a remote *Player*. They can use the "Archive Me" command to archive the current screen to a H.264 movie. The users can also change the *Streamer*'s name attribute.

2.1.2 Player

The *Player* uses Zlib to uncompress the stream. It then reverses the byte-map transformation to retrieve the screen update. The update is then painted in the receiving screen.

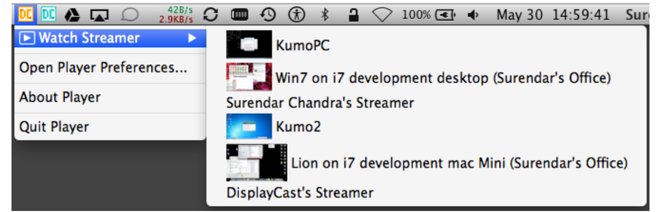
Users interact with the *Player* using a task bar system tray interface (Figure 3). Users can use the "Watch Streamer" command to monitor specific *Streamers*. On the Mac OS X, the user interface displays a thumb nail snapshot of each Streamer to aid in choosing the correct Streamer (Figure 3(a)).

2.1.3 Archiver

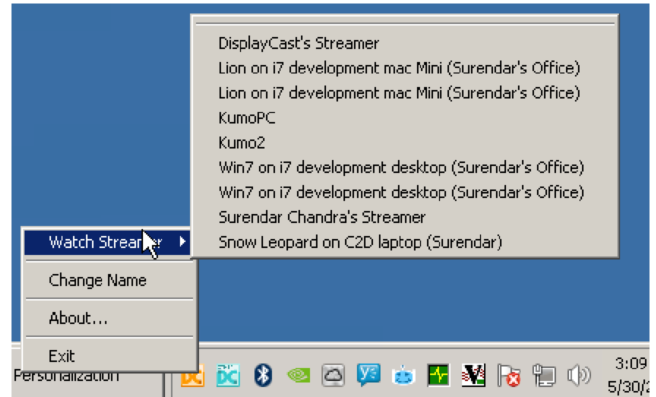
The *Archiver* decodes the compressed updates. It then uses the *AV Foundation* framework that was introduced in Mac OS X 10.7 to create a H.264 video of the streamed screen updates. We attempt to create a 60 fps video. The actual video rate depends on the available computing resources. Using the latest Intel core i5/i7 processors that use the HD 3000 GPU and a H.264 encoder, we achieved about 20 fps for some scenarios. More importantly, when no screen updates are received, no updates are sent to the compressor. Thus, during inactive durations, we achieve higher compression ratios than is achieved by systems that treat screens as a fixed frame rate movies (e.g., Quicktime screen recorder). During active duration spikes, we achieve higher frame rate than prior approaches (Quicktime achieved about 14 fps).

2.2 Naming and location

DisplayCast components use Zeroconf [9] to advertise and locate each other. Zeroconf uses link-local multicast for service advertisement. It can also use wide-area DNS to locate objects on a global



(a) OS X (displays a snapshot of Streamers)



(b) Windows 7

Figure 3: Player system tray user interface

scale. Wide-area DNS servers can be configured with modest effort; the challenge is in managing the keys required for a secure service resource record updation.

We used Bonjour: Apple's implementation of Zeroconf. Each entity (*Streamer*, *Player* and *Archiver*) generates a 128 bit globally unique identifier to identify itself. We disclose attributes of each stream and the capabilities of the receiver using the Zeroconf TXT records. A user configurable name is an attribute that is advertised by all the services. *Streamers* disclose the *userid*, *osVersion*, image *snapshot* port, *screen* size and a list of nearby *Players*. *Players* disclose their bluetooth ID so that *Streamers* can identify nearby *Players* using a bluetooth scan operation. *Players* also enumerate the window parameters of each active session.

Bonjour uses an asynchronous API to advertise and locate other services. We include a Windows 7 service that collects these asynchronous notifications and provides an synchronous mechanism to locate and access the various components using a HTTP/REST service. This service responds to queries using JSONP. The *Streamers*, *Archivers* and *Players* provide a HTTP/REST service for remote command initiation. The specific ports used for these services are advertised through the Bonjour TXT records. It is expected that application developers will access these services using the synchronous service controller. The API for this service is described in the source repository. For example, the following HTTP/REST command (`http://displaycast.fxpal.net:11223/connect?source=Streamer&sink=Player&callback=jsonID`) sent to the API controller on host display.fxpal.net initiates a new session between a source *Streamer* and a sink *Player*. A session identifier is returned (via a JSONP response) that can be used for placement control on the *Player*.

3. GETTING AND BUILDING DISPLAYCAST

DisplayCast is released under a New BSD License. The specific license terms are included along with the source. Both the Mac

(and the iOS Player) and the Windows software are released as separate git repositories in the DisplayCast github project (https://github.com/DisplayCast). A zip archive of the latest source code is available online for each repository. Prebuilt binaries are also available in the download area. Wiki for each repository describes the build instructions in further detail. Next, we describe the build requirements for these two projects in further detail.

3.1 Apple repository

The Mac and IOS git repository is available at https://github.com/DisplayCast/OSX-IOS/. A zip archive is available at https://github.com/DisplayCast/OSX-IOS/zipball/master. We used Xcode 4.4 for developing the software. We provide a *DisplayCast* workspace which contains *OSX* and *iOS* projects. The *iOS* project defines the *Player* target while the *OSX* project defines five targets: *Player*, *Streamer*, *Archiver* and *Preferences*. We also provide a virtual target called *All* that helps compile all the targets in a single operation. The workspace is configured to use any installed development or deployment certificates (required for submission to the Apple Appstore and enforced by gatekeeper in Mac OS X 10.8+).

We provide an utility shell script *OSX/Packager/pkmaker.sh* to generate an installer package, wrapped inside a disk-image. The disk-image is available in the *Installers* directory and includes an Applescript called *EnableAutoLogin* that configures the system to automatically start the *Streamer* and *Player* on a system reboot. The Xcode packagemaker tool appears to be deprecated from Xcode 4.4. Hence we used Packages tool available at http://s.sudre.free.fr/Software/Packages/about.html.

3.2 Windows repository

The Windows 7 project is available at https://github.com/DisplayCast/Win7. A zip archive of the entire repository is available at https://github.com/DisplayCast/Win7/zipball/master. We used Visual Studio 2010 Professional to develop this project. The free Visual Studio 2010 Express can also be used for compiling the project though the free tool does not allow one to create the installers.

We require the following three components to be pre-installed:

- Demoforge Mirror Driver: We used the Demoforge mirror driver that is available at http://www.demoforge.com/dfmirage.htm. We used the MirrSharp code available at https://code.google.com/p/cellbank/ as the basis for interfacing our C# code with the mirror driver.
- Apple Bonjour: We used the Bonjour Zero-Configuration SDK available at https://developer.apple.com/opensource/. The c# wrapper (http://code.google.com/p/zeroconfignetservices/) with all relevant patches is already incorporated into DisplayCast.
- 32feet.NET: Available at http://32feet.codeplex.com, this software provides an easy way to access Bluetooth functionality from C#.

The *DisplayCast* solution consists of the *ControlService*, *Stream* and *Player* project. Support for Cisco WiFi localization is provided by the *Location* project, while *Zeroconfservice* provides the C# interface to Bonjour. The installer projects, *DisplayCast Installer* and *ControllerService Installer* can be built separately to generate the Windows installers for the DisplayCast system and the service that provides a synchronous HTTP/REST service to the DisplayCast system. The API for the Controller Service is available in the *Documentation/Control Service API.txt* file. Note that the *Streamer* requires administrative privileges; Visual Studio can be started as an administrator to ease the debugging process.

4. USING DISPLAYCAST

The *Streamer* and *Player* applications are installed in the system *Application/FXPAL* folder. On a Mac, the *Archiver* application is also installed in this folder. The *Streamer* and *Player* applications are automatically started on a system reboot.

5. CONCLUSION

Prior screen sharing systems did not provide adequate performance for use in Intranet scenarios. DisplayCast achieves a seamless performance for a variety of application scenarios. Our implementation natively works on Windows and Mac OS X.

Linux uses network optimized X windows. X applications pre-render their screen to a pixmap and send the final output to the screen. Further analysis of Linux screen contents is necessary to develop a high performance Streamer for Linux. Porting the Streamer to Android or iOS devices requires rooting and jail breaking in order to circumvent their built in restrictions. Developed of *Player* for Linux and HTML5 should be relatively straightforward. Future work will introduce a global naming and location mechanism, integration with LDAP for authentication and authorization as well as allow remote players to interact with the host applications.

Acknowledgements

Tony Dunnigan designed the logos used by DisplayCast.

6. REFERENCES

[1] Demoforge mirage driver (dfmirage video hook driver). http://www.demoforge.com/dfmirage.htm.

[2] Stream movies and music wirelessly with airplay. http://www.apple.com/ipad/features/airplay/.

[3] CHANDRA, S., BIEHL, J. T., BORECZKY, J., CARTER, S., AND ROWE, L. A. Understanding screen contents for building a high performance, real time screen sharing system. In *ACM Multimedia 2012* (Nara, Japan, Oct. 2012).

[4] LOUP GAILLY, J., AND ADLER, M. zlib: A massively spiffy yet delicately unobtrusive compression library. zlib.net.

[5] RICHARDSON, T., AND LEVINE, J. The remote framebuffer protocol. RFC 6143, Mar. 2011.

[6] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing 2* (Jan. 1998), 33–38.

[7] VON HOFFMAN, J. T. *Guide to Distributed PowerPoint*. Boston University, 2001.

[8] WALLACE, G., AND LI, K. Virtually shared displays and user input devices. In *USENIX Annual Technical Conference '07* (Santa Clara, CA, USA, June 2007), pp. 375–380.

[9] Zero configuration networking (zeroconf). http://www.zeroconf.org/.