Characterising Sun Ray™ Thin Client Performance

Richard Smith

Sun Microsystems

<richard.smith@sun.com>

ABSTRACT

Thin clients have different performance characteristics to traditional "fat" clients, and there are workloads with which they may struggle. Added to the mix can be many layers of software, each with its own set of performance foibles. Since thin clients are often deployed in situations where Total Cost of Ownership [TCO] is a significant factor, a key question is how many users can a given solution support.

The paper discusses modelling the performance of Sun Microsystems' Sun Ray™ thin client solution for various workloads, and identifies some of the significant factors to be incorporated in sizing and capacity planning.

1. Introduction

Sun Ray thin-client appliance (desktop unit or DTU) and associated server software provides an alternative to traditional fat clients for desktop computing. In this architecture, the DTU provides display services and typical human input devices, such as keyboard and mouse, through which a user accesses their applications. Other than its ethernet address it is stateless. All state information and application functionality is kept on an associated, centrally managed server.

Communication between the server and DTU is via a set of proprietary protocols implemented on top of UDP, generally on a private ethernet network. The protocols provide for session management, device management, rendering and interaction, along with adding reliability to UDP.

A key part of the architecture is that each session is associated with its own X server. X servers generally have a device independent part, DIX, and a device dependent part, DDX. On Solaris, `/usr/openwin/bin/Xsun` provides the DIX part, and dynamically loads shared libraries that implement the DDX part. For Sun Ray the DDX part is `/usr/openwin/server/modules/ddxSUNWsunray.so.1`. This converts between X11 and Sun Ray protocols. Internally it maintains a virtual frame buffer and maps changes to the contents into Sun Ray protocol such that what is displayed mirrors the virtual frame buffer. Incoming mouse and keyboard events form a DTU have to be converted into X events. In this way a user interacts, via a DTU and X server, with a set of X applications.

The applications themselves need not be local to the Sun Ray server. They can be remote X applications (since X is inherently network-based), or alternatively remote desktop protocols like RDP, VNC or Citrix ICA can be used to access both X- and non-X applications on remote servers.

This architecture has several potential benefits for an enterprise:

- Lower administrative cost. Administration is centralised, with far fewer servers to be maintained than the number of DTUs supported.
- No state information is kept on the DTU so there is nothing to be installed and no data that can be lost if the DTU dies.
- Through the Hot Desk capability, user sessions can be moved from one DTU to another without requiring an intermediate logoff and logon. The sessions continue to exist when unbound from a DTU.
- Security is enhanced since there is no data kept locally to be misappropriated, and with recent versions of the server software and appliance firmware data can be strongly encrypted before transmission. Sun Ray also has a builtin smartcard reader that can be used for smartcard/token-based authentication.
- The appliances themselves can be relatively low-power devices that require no fan and hence are very quiet. Other than the display device, this means they can be quite compact.
- Firmware updates are transparently delivered to Sun Ray appliances on power-up, or can be forced by administrative command. Improvements can be delivered without a massive rollout project.

However there are performance implications for users. They no longer have dedicated hardware on their desktop running applications locally and capable of writing directly to frame buffer memory. This has the following consequences:

- There is additional latency in user interaction, such as responding to a button press.
- DDX protocol conversion can use a non-trivial amount of cpu time, especially for high pixel-rate rendering.
- Since resources are shared, there is a non-zero probability of contention. Contention increases latency beyond what a user of a lightly loaded system would experience.

So Sun Ray represents a tradeoff between the benefits and performance implications. Resource sharing is interesting because of the extremely poor utilisation of the typical desktop PC in a large organisation. Experience has shown that these organisations rarely know what the average utilisation of all their servers are, but when it is measured, its down around 10%, with PCs at least an order of magnitude worse. Part of the skill in a large Sun Ray deployment then is to strike the right balance between acceptable performance and higher resource utilisation.

2. Case Study

The motivation for this paper came from the author's experiences in investigating performance issues with a particular deployment, and subsequent experiments in a more controlled environment.

The applications to be delivered were typical office productivity software such as Microsoft Office97, Internet Explorer, and Adobe Acroread running on a farm of Windows servers, accessed via Citrix ICA client on the Sun Ray server.

Typical usage patterns were developed, although not rigorously validated against actual user usage. From testing, several performance issues were identified and roughly categorised. They were:

- Slow scrolling.
- Screen overruns, where scrolling or highlighting failed to respond rapidly to mouse button release.
- Stutter and slowness in PowerPoint animations.
- Lag on mouseover.

The customer's acceptable performance criteria included terms such as “smooth scrolling”, “zero stutter”, “no overrun”, and “instant update”. Its not possible to do anything in zero time, nor is it possible to scroll things more smoothly than a monitor’s refresh rate permits, so some latitude needs to be given when interpreting the criteria.

One cause of overruns was quickly identified as a bug in the Citrix ICA client v7.08. This manifested itself as a delay of over 5 seconds from receiving an `XButtonReleasedEvent` before passing on the equivalent event in ICA protocol. Moving to ICA v7.19 seemed to fix this.

Another source of overruns was identified as a bug in Office97, which is fixed in Office2003. The problem also showed up when not using a Sun Ray, which is a timely reminder that performance problems can appear anywhere in an overall solution, and it can take a careful choice of a series of experiments to identify root cause. It was certainly unfair to blame Sun Ray for this one.

One common problem was the impact of scrolling on performance. Scrolling is an example of a high-pixel rate activity, in which a large number of pixels on DTU's display screen have to be changed repeatedly, many times a second. As discussed in subsequent sections, the current implementation is likely to develop a bottleneck around the computation of changes to regions of display space and transferring pixel update data to the DTU.

A distinction needs to be made between what an architecture imposes and what an actual implementation does. There is room for more efficient implementations of a Sun Ray X server. Research continues in SunLabs and elsewhere about mapping X protocol streams into Sun Ray protocol to reduce bandwidth or make more use of hardware capabilities. Since the initial investigation into performance at the customer site, there have been several improvements shipped as patches to the Sun Ray Server 2.0 software.

3. Monitoring and Measurement

Understanding Sun Ray performance involves building a picture of where time is being spent and why. Its unrealistic to expect a single tool to provide all the answers, and its pretty hard to identify up front all the data that may be required. In some sense its like detective work, where evidence is collected which in turn suggests further lines of enquiry.

The following are some tools and data sources that may be of use when analysing Sun Ray performance.

3.1 prstat -mL

The `prstat` utility examines all active processes on the system and reports selected statistics. The `-m` option enables microstate accounting for each process, so that Solaris accurately measures the time spent in each microstate (user cpu, system cpu, wait for cpu, asleep, ...). Without microstate accounting, some of the buckets are not maintained, in particular wait for cpu time. Only the owner of a process or superuser can turn on/off microstate accounting.

The `-L` option tells `prstat` to report on each LightWeight Process [LWP] rather than at a process level. An LWP is a unit of concurrency, and can only use one cpu at a time. It can only be in one state at a time, so that the sum of time spent in each state should be 100%. A single-threaded process has only one LWP, whereas multithreaded processes usually have more than one. Aggregate statistics for a multithreaded process can be confusing compared to a set of LWP statistics.

`prstat` has some weaknesses when trying to get a picture of a system with a large number of processes or LWPs:

- It rounds the data to relatively low precision.
- Its calculation of percentages is based on process existence rather than width of observation interval.
- Short-lived processes may not be captured.

It therefore can be useful to write a supplementary tool to display data from `/proc` pseudo filesystem at higher resolution, and include child cpu time buckets. When a process dies its cpu time is accumulated in the child cpu time buckets of its parent.

3.2 pstack

Taking a snapshot of a process' stack gives clues about where time is spent, as it shows not just what is currently being executed but also all the functions that directly or indirectly have called it. Function names are often quite meaningful or suggestive of their purpose. LWPs that are asleep will often be inside `poll()` or `lwp_park()`.

3.3 performance data collector

A particularly useful tool for profiling processes is the performance data collector and analyser, the latest version of which is part of Sun Studio 9. This takes regular snapshots of process stacks (which includes LWP stacks for multithreaded processes), so that a statistical profile of where time is being spent can be built. Each snapshot includes what the call stack looks like, and what the current microstate is. A timeline view can be displayed via the analyzer GUI.

Unfortunately the author has had little success in dynamically attaching to `Xsun` processes. For profiling purposes on a standalone system, one possibility is to edit temporarily the `/opt/SUNWut/lib/utxsun` script, which launches Sun Ray `Xsun` processes when a user logs on.

3.4 interpose library

Shared libraries provide implementations of APIs that a program can use without physically including the functions in its executable text. At runtime the loader dynamically loads shared libraries into an address space as required. The `ldd` command shows the library dependencies for a program, and `pldd` command shows the dependencies for an executing process.

Through the `LD_PRELOAD` mechanism, an alternate implementation of selected functions can be substituted for the originals. The alternate implementation is still able to call the original functions, but can do many other useful things first, such as modify parameters, log details of the call to a file, or measure the time spent in each call.

3.5 truss -u

Recent versions of `truss` have added a builtin interpose capability for shared libraries that can be used to time entry and exit of functions. For example calls to `newtCopyArea()` can be logged:

```
truss -t!all -s!all -uddxSUNWsunray:newtCopyArea -p pid
```

This can be fairly heavyweight so care needs to be taken about how many function calls are being interposed per second to minimise performance degradation.

3.6 lockstat

The `lockstat` utility gathers and displays kernel locking and profiling statistics. It can be used to identify contention for kernel resources, such as over a per-file POSIX RW lock. Another, less obvious use, is to break down where system cpu time is being consumed in the kernel. While `truss -c` can be used on a process (with care, since it does add some overhead), `lockstat` can be used to get a system-wide view. It can also be used to probe deeper than a system call, such as working out how much cpu time consumed by `read()` is due to disk i/o and how much is network i/o.

Solaris 10 introduces a new capability, DTrace, which is likely to become the tool of

choice in the future when analysing kernel activity.

3.7 kstat and SE Toolkit

Utilities such as `vmstat`, `mpstat`, `iostat` and `netstat` have as their common source of information the collection of performance data maintained by the kstat facility. kstat was created to avoid the security and maintenance problems associated with commands having direct access to kernel memory. The contents of kstat can be collectively dumped via the `kstat(1M)` command.

The SE Toolkit [www.setoolkit.com] uses a scripting language to access and display useful information from `kstat` and other sources. For TCP network traffic, the author has found the `netstat.se` script useful. The current Sun Ray protocol implementation though uses UDP, for which `netstat.se` is inadequate. However its not hard to clone a new version based on `netstat.se` that displays octets in and out per interface. This of course includes UDP traffic.

3.8 utcapture

The `utcapture -r` tool collects data about activity to and from Sun Rays every 15 seconds. This includes packets sent and dropped, round trip latency, and the number of bytes sent. Bytes sent strongly correlates with cpu load on the Sun Ray server.

3.9 /var/dt/st.n

Recent patches to Sun Ray Server 2.0 software have added an undocumented file per Sun Ray that is updated every second with performance information. While some of the data is cryptic, others can be identified with various kinds of display function such as the Sun Ray equivalent of an `XCopyArea()` from one region of display memory to another.

3.10 pmap -x

Modern unices make heavy use of shared libraries and Copy On Write [COW] to share memory and reduce per instance memory usage. Memory usage of a particular process can be analysed via the `pmap -x` command, provided care is taken to distinguish between shared and private memory. Running out of main memory can be diabolical for good interactive performance, so the page scan rate (such as from `vmstat` or `sar`) should be closely monitored in a Sun Ray environment if it is much above zero.

3.11 android

Android is an open-source tool for recording and playing back scripts of X11 interactions. The initial capture is done by using an `xscope` utility to interpose on X11 events being sent to an application and placing details of keyboard and mouse activity into a file. The script can then be edited, cleaned up, modifed, and ultimately played back through an X server via the `XTest` extension. The synthetic user still needs to have authority to connect to the Xserver, which in some cases may involve copying a user's `.Xauthority` file and using `XAUTHORITY` environment variable to point to it.

This is one way of simulating a Sun Ray user, although it is isn't an exact workload match since the input event processing doesn't occur in the same DDX module as a real user would. However since rendering tends to dominate cpu usage its a reasonable approximation.

4. Scrolling on Sun Ray

Since issues associated with scrolling seemed to dominate the list of problems in the deployment discussed previously, a special focus was placed on how scrolling performed. An X program was written that scrolled an image endlessly, and provided a reproduceable workload. The program had parameters to control the window size, pause between scrolls, and number of rows of pixels to scroll down each scroll.

Scrolling down involves logically shifting window contents up one or more rows of pixels and rendering (painting) new rows of pixels at the bottom. Just as in the movies, this *could* be done by rendering an entire new frame each scroll. However this would consume a lot of unnecessary bandwith. In X applications an off-screen image [pixmap] can be created in the X server from which rectangular regions of pixels can be copied to the display screen. This dramatically reduces the bandwidth between an X application [client] and X server.

Infact a pixmap and window are both objects of Drawable type. X permits copying from one region of a window to another even when the regions overlap. Doing so saves even further bandwidth, this time between the X server and display device. In the case of Sun Ray appliances, the display memory is really only sufficient for the screen. Off-screen pixmaps are held back at the X server. When scrolling by copying from a pixmap, every pixel in the window changes, and the current implementation generates network traffic proportional to the window size and scroll rate. Worse, the analysis of changes to the display is naive and consumes a lot of cpu time. The network to the Sun Ray is easily saturated, transmission of pixels dominates, and

interactive reponse rapidly deteriorates. It is therefore very desirable to maximise copying in screen space, such as within a window.

One of the complaints about scrolling performance on Sun Ray was the scroll rate, the rate at which rows of pixels are moved up (or down) the screen. Modern display devices typically have a refresh rate between 60 and 78Hz, so it is not possible to display distinct images faster than that. What this means is that scrolling one row of pixels at a time per refresh looks very smooth but is too slow for a typical user moving around a document. It might take 10 seconds to move from one screenful to the next. Instead, it is necessary to shift the window by multiple lines per scroll. Using the interpose library in Appendix A, it was observed that acroread v5.08 shifted by 16 lines each scroll, at a rate of about 20 scrolls per second.

In Table 1 two versions of a scrolling program are compared for different window sizes. One program uses a window-to-window copy, and the other a pixmap-to-window copy. When the system is allowed to scroll as fast as it can the cpu time is naturally high, but the difference in scroll rate between the two is dramatic! It actually makes little sense to render faster than the screen refresh rate, but a slow scroll increases the likelihood of visual artifacts being apparent from the failure to synchronise screen copying with the screen's VBLANK cycle. OpenGL implementations [3D graphics] generally use double buffering and synchronise buffer switching with VBLANK to avoid this.

Taking deltas from a series of `/var/dt/st.`*n* files at 1 second intervals gives more detail about what is happening. Table 2 compares 1000x700 pause=10ms scroll=1 for the two programs.

From this it appears that `pixels` counts the number of pixels being changed on the screen per second; `bytes` is a measure of bandwidth used to the device; `set` counts new pixels sent to the device (and each pixel takes 3 bytes for 24-bit colour); and `cpy` counts how many pixels were changed within the display device via a screen-to-screen copy. The difference in pixels per second for the two programs is huge: 64.4 MB/s vs 2.7MB/s. Clearly finding ways of maximising `cpy` over `set` is important for good performance.

A quick check of rdesktop 1.3.1 via the interpose library showed that it was using significant screen to screen copies and with the current Sun Ray stack it performed well when scrolling in Excel:

```
  ms      src     dest srcx srcy    w    h dstx dsty
---- -------- -------- ---- ---- ---- ---- ---- ----
```

pause (ms)	scroll	width	height	screen to screen				pixmap to screen			
				scroll/s	usr	sys	MB/s	scroll/s	usr	sys	MB/s
0	1	1000	100	980	68	9	3.2	29	14	16	8.9
0	1	1000	200	568	74	6	1.9	16	16	14	8.9
0	1	1000	300	397	77	4	1.3	10	15	14	8.9
0	1	1000	400	304	76	3	1.0	7	16	14	8.6
0	1	1000	500	250	78	3	0.8	6	15	14	8.3
0	1	1000	600	207	77	3	0.7	5	15	14	8.2
0	1	1000	700	181	79	2	0.6	5	16	14	8.4
0	20	1000	700	90	47	10	5.7	5	16	14	8.4
10	1	1000	700	92	37	1	0.3	5	16	13	8.3
10	20	1000	700	92	49	11	5.9	5	16	14	8.5

Table 1: Two versions of a scrolling program are compared for different window sizes.

	time	tick	pixels	bytes	set	cpy
screen to screen	1.0	51	64,403,561	285,812	82,524	64,777,372
	1.0	50	64,403,505	285,664	82,524	64,785,468
	1.0	50	63,704,089	281,972	84,617	63,321,432
	1.0	50	64,403,535	285,616	82,823	64,801,524
pixmap to screen	1.0	50	2,733,073	8,286,672	2,478,168	0
	1.0	50	2,769,279	8,384,124	2,522,570	0
	1.0	52	2,634,122	7,985,968	2,385,788	0
	1.0	50	2,682,293	8,140,188	2,443,898	0

Table 2: Comparison of 1000x700 pause=10ms scroll=1 for the two programs.

```
  0 00700002 00700002    2  231  749  300    2  143
 39 00700002 00700002    2  231  749  300    2  143
  0 00700002 00700002    2  231  749  300    2  143
  0 00700002 00700002    2  231  749  300    2  143
 30 00700002 00700002    2  231  749  300    2  143
  0 00700002 00700002    2  231  749  300    2  143
```

5. Conclusions and Future Work

Sun Rays can perform well for applications that mostly paint forms. High pixel rate applications, such as for scrolling, are more challenging but can still perform acceptably if the majority of pixels can be rendered via screen-to-screen copying. This is determined by the nature of applications and how they are written.

There is scope for future implementations of Sun Ray X servers to be more intelligent in mapping X11 protocol to Sun Ray protocol. By tracking regions of pixmaps copied to the display screen, it is in principle possible to make greater use of the much more efficient screen-to-screen copy. Another strategy is to reduce the number of “frames” rendered per second dynamically. The cpu cost of rendering to the virtual frame buffer is much less than the cost of analysing and sending pixels to the DTU. Accumulating changes in the virtual frame buffer before updating the screen could lower the cpu cost, but add latency.

Outside of the Sun Ray system itself, additional protocols may be involved and which also need to be managed and measured. These include various remote desktop protocols. Future work would be to extend the analysis of where time is spent to incorporate the end-to-end response time of a deployment using these protocols.

Appendix A

```
/*
 * Example interpose library--to compile:
 *
 * cc -I/usr/openwin/share/include/X11 -Kpic -G \
 *     interpose.c -o libinterpose.so
 * LD_PRELOAD=./libinterpose.so cmd ...
 */

#include <X11/Xlib.h>
#include <stdio.h>
#include <stddef.h>
#include <dlfcn.h>
#include <time.h>

static int (*xcpy_handle)(Display *, Drawable, Drawable, GC,
int, int, unsigned int, unsigned int, int, int) = NULL;
static hrtime_t t1, t2;
static int nlines = 0;

#pragma init(initptrs)

void initptrs(void)
{
xcpy_handle = (int (*)(Display *, Drawable, Drawable, GC,
int, int, unsigned int, unsigned int, int, int))
dlsym(RTLD_NEXT, "XCopyArea");
t1 = gethrtime();
}

static void header(void)
{
printf("    ms      src     dest srcx srcy    w    h dstx dsty\n");
printf("------ -------- -------- ---- ---- ---- ---- ---- ----\n");
}

int XCopyArea(Display *display, Drawable src, Drawable dest,
GC gc, int srcx, int srcy, unsigned int w, unsigned int h,
int dstx, int dsty)
{
if (nlines == 0)
header();
nlines++;
if (nlines >= 20)
nlines = 0;
t2 = gethrtime();
printf("%6.0f %8.8x %8.8x %4d %4d %4u %4u %4d %4d\n",
(t2 - t1)*1e-6,
src, dest, srcx, srcy, w, h, dstx, dsty);
t1 = t2;
return xcpy_handle(display, src, dest, gc, srcx, srcy, w, h, dstx, dsty);
}
```