Improving VNC Performance

Cynthia Taylor and Joe Pasquale
Department of Computer Science & Engineering
University of California, San Diego
{sriram,pasquale}@cs.ucsd.edu

ABSTRACT
Virtual Network Computing, or VNC, is a popular thin client application used to access files and applications on remote computers. It is especially relevant as infrastructure to support ubiquitous computing applications, as it offers a way to run data-and-computation-intensive applications and allow users to access them through lightweight devices. However, VNC can suffer from significant losses in throughput when there is high latency between the client and server.

In this work, we present a *Message Accelerator* proxy for VNC. This Message Accelerator mitigates high latency network effects while maintaining the advantages of a client-pull system. By operating near/on the server, it can send updates to the client at a rate corresponding to proxy-server interactions which are faster than client-server interactions. When testing using video, our Message Accelerator design results in frame rates an order of magnitude higher than plain VNC when running under high latency conditions.

Author Keywords
VNC, remote desktop control, thin client, distributed computing

ACM Classification Keywords
D.0 Software: General

INTRODUCTION
Ubiquitous computing is characterized by context-aware applications that fit unobtrusively into the user's life. These applications are frequently aware of the user's location, what is happening visually around the user, or other details of the user's changing surroundings. Effectively processing these details will eventually depend on filtering large amounts of data through computationally intensive machine learning and computer vision applications. However, for devices to comfortably travel with the user, they must be at least as small as a PDA or a cellphone, and ideally be something the user would normally carry through their daily life. While these small devices offer the ability to fully integrate into the user's life regardless of location, they generally will not have enough computing power to take advantage of it by running context-aware applications.

Thin client systems such as VNC allow client devices with very little computational power to run computationally demanding processes remotely, while creating the illusion that they are running locally. They allow long running processes to execute on servers that reboot less frequently than a user's personal computer or devices that may run out of battery energy. While thin clients have existed in different forms for many decades, trends in computer use today indicate that now is an especially apt time for them [7]. For example, in business environments, the client devices can have very little computational power, allowing businesses to use cheap terminals instead of expensive PCs. Having all of the data and computation running on a central server means that a business can just update software on one computer instead of hundreds or thousands of PCs running on employees' desks, saving money on maintenance. Keeping all data on a central server provides security from lost laptops or disgruntled employees. Even the electricity costs of running thin clients can be cheaper than that of running PCs. As a result, many businesses have been moving over to thin client systems [6, 18].

However, one problem with thin client systems is that they may suffer from poor performance, resulting from a number of factors. There is the computational overhead on the server side of tracking and encoding display updates, and of decoding and rendering them on the client side. There is the limiting factor of bandwidth, or how much data can be sent across the network in a given time period. Perhaps the most significant limiting factor in thin client systems is network latency [5]. Latency affects every message, regardless of its size, sent between the client and server, so it is important that it be minimized. These performance effects are especially noticeable in media-intensive applications, such as video.

Several ubiquitous computing projects are already using VNC as a component of supporting infrastructure in their systems, either using available implementations as they are, or modifying them to suit their needs [17, 8, 4, 3]. However, the poor performance that VNC demonstrates under latency conditions has prevented it from being as successful in projects where the user may be a significant distance from the VNC server. For example, the Labscape project by Arnstein et al attempted an early implementation in which they used VNC, but had to scrap it because their users were unhappy

with VNC's responsiveness [1]. Takashio et al, in their goal to support "Follow-Me applications in ubiquitous computing environments," note the value of realizing user mobility by using VNC, and yet they explicitly reject it because of networking problems, both bandwidth limitations and latency concerns, and thus propose a new mobile agent framework [15]. Unfortunately, this also sacrifices the use of VNC as a standard, well accepted, and highly deployed system, simply because of its poor performance in particular situations. In this work, we offer a way to mitigate VNC's poor performance under high latency conditions, making it a more flexible and useful tool for ubiquitous computing systems.

The solution that we present is a very general and simple proxy, called a *Message Accelerator*, that works with VNC to mitigate the effects of network latency. Our Message Accelerator runs on or near the server machine and requests updates from the server in a client-pull fashion, thus mimicking the VNC client. It then forwards these updates to the client as soon as possible, at the faster rate of proxy-server interactions rather than the slower rate of client-server interactions. Because the Message Accelerator does not wait to receive a request from the client before sending, it is not affected by high network latency in the same way that traditional client-pull systems are affected. Our Message Accelerator requires no changes to the existing client or server, making it easy for a user to install (essentially "plug and play"), and eliminating parallel code maintenance issues. Using our Message Accelerator with VNC can result in the client receiving updates an order of magnitude faster in high latency situations.

We will first provide some background and related work in Section 2. In Section 3, we will go into more detail about VNC. In Section 4, we will describe our changes to VNC. In Section 5, we will discuss our experiments and results. In Section 6 we describe how our system evolved from our original design to the system we present here. We will briefly discuss future works in Section 7, and then conclude in Section 8.

BACKGROUND AND RELATED WORK

This work is an adaption of the Virtual Network Computing or VNC system [13]. The VNC system works as a user-space application that requires no modifications to either the operating system or user applications. The VNC server continually scrapes the framebuffer, an area of memory in which color values for every on-screen pixel are stored. The VNC client sends a request about a specific on-screen rectangle, and the server responds with an update consisting of an encoding of the changes between now and the last time the client requested information about that rectangle. We will go into more detail about VNC in Section 3.

Thin client systems fall into two general categories in terms of when the server sends an update to the client. In *server-push* systems, updates are sent to the client as soon as the server generates them. For example, if the user is playing video, in a server-push system, when a new frame of video is played, the server will immediately send an update to the client with the changes to the framebuffer. In *client-pull* systems, the server waits to send an update until it receives an explicit request from the client. If a user is playing video on a client-pull system, when a new frame of video is available or is generated, the server stores the changes it makes to the framebuffer in an internal representation, and when the server receives a request from the client, it sends an update containing all the changes to the framebuffer since the last update. VNC is a client-pull system.

The SLIM system offers a contrast to VNC [14]. SLIM uses a very similar message protocol to VNC, but differs greatly in other ways. Instead of scraping the framebuffer to detect updates, SLIM works as a virtual device driver and offers a software library with display options for higher-level applications. In SLIM's implementation, X-Windows is modified to use the SLIM virtual device driver so that applications can either explicitly invoke SLIM display commands through its library, or pass their display commands to X-windows and have it forward them to SLIM. In this way, applications with specific display needs such as multimedia applications can send commands to SLIM that include more semantic information about how they should be processed. However, applications must be specifically altered to take advantage of this. Even if no user-level applications are modified, at the very least the window manager must be modified, and the modified applications must be kept updated in parallel with the original applications. The system is also limited to operating systems with available source code because of this. SLIM uses a server-push update system, and requires a network and machines capable of handling server-push updates with no loss of data.

THINC also acts as a virtual device driver, but at the abstract hardware level so no changes need to be made to applications or operating system [2]. THINC also uses a similar message protocol to SLIM and VNC. Device driver commands are tracked in an intermediate encoding until they are finally translated into messages and sent to the client. Commands are highly processed while in their intermediate encoding to avoid sending any information that is fully or partially overwritten by later updates. THINC is a server-push system, but rather than sending messages immediately, before sending a message to the client it checks the socket to see if the write will block, and waits until the buffer is next flushed on blocking. THINC uses special handling for video, using a stream-based system to avoid jitter between frames. The special handling of video adds even more complexity to an already quite complex system. In addition, since device drivers are OS-specific, there must be a virtual device driver written for each operating system on which THINC is used.

This work was also informed by the papers of Nieh et al, which describe and compare many currently existing thin client systems [10, 11, 5]. Nieh stresses that the performance of thin client systems is especially affected by latency, rather than bandwidth, and that client-pull systems are affected more than others due to the fact that they must send two messages for each update.

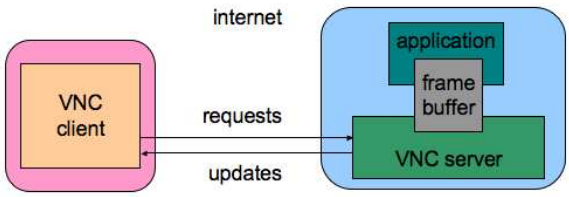

Figure 1. The VNC server scans the framebuffer and sends updates to the client.

DETAILS OF VNC

In this section, we will describe the basic workings of a particular implementation of VNC, TightVNC [16]. These details can easily be generalized to other VNC systems. We explain the concept of client-pull, the core operating feature of this and all VNC systems, in the context of a real working system that is publicly available and in wide use. We will focus on the implementation of the RFB protocol and the sending of messages between the client and server [12]. The system is illustrated in Figure 1.

The VNC Client

The VNC client is implemented as a very simple program; it is not multithreaded, and it blocks on reads and writes. After a brief initialization phase in which it exchanges setup information with the server, it falls into a while(true) loop. In this loop, it first waits to receive an update from the server. It processes the update and redraws the display. Then it issues a request for a new update.

The client intersperses read calls throughout the processing of the update, rather than reading the entire update and then processing it. The structure of the update encourages this: it contains a general header containing information about the message, and then a series of rectangles, each with a rectangle header that contains the dimensions of the data following it and its encoding. Due to this message structure, it is impossible to tell how many bytes an update is without processing a significant portion of the message. An additional advantage is that by processing the message as it is being read, the client can get as much processing as possible done before it has received the message in its entirety.

There are times when the client has processed as much of the update as it has received, but cannot read any more from the server. When this happens, the client uses the idle time to collect user input, such as mouse movement and typing, and sends the appropriate messages to the server. This is the only time when it processes user input.

The VNC Server

The VNC server is more complex than the client. As the server runs, it notes when the framebuffer is changed. It stores an internal representation of the area modified, and the new modifications made to it. This is called the *modified region*. The server continues to add any modifications made to the modified region until it receives a request from the client.

The requests the client sends are very short, containing just the dimensions of a rectangle, and a bit specifying whether the update is incremental or not. The rectangle is always the dimensions of the client's display window and rarely changes within a session. The first request the client sends is not incremental, and all requests afterwards are incremental. If the request the server receives is not incremental, the server immediately sends all of the framebuffer information for the rectangle.

If the request is incremental, the server compares the rectangle the client requested to the modified region. If they overlap, the server sends an update with the modifications within the overlapping area, and clears those modifications from the modified region. If the areas do not overlap, as in the case where no changes have been made to the framebuffer since the last request, the server saves the rectangle as the *requested region*. The next time the framebuffer is modified, the server checks to see if the modification falls within the requested region. If it does, the server sends the client an update with the new modifications and deletes the requested region. If it does not, the server adds the modifications to the modified region and keeps the requested region. This means that the server can send an update in response to a request from the client any time after it receives the request. If the server receives more than one request before there are any modifications to the requested region, it will just send one update in response to all requests. We emphasize though that the server will only send an update if it receives a request from the client, and no sooner than having received the request.

VNC as a Client-Pull System

As described above, VNC is a client-pull system. This has both advantages and disadvantages in terms of system performance. Having the server wait until it receives a request to send an update provides a very simple flow-control mechanism to prevent the client from being overwhelmed by updates. The client will never receive an update that it is not ready to process, since the server sends updates only after the client specifically requests them. This is especially good for the types of highly resource-limited clients that motivate this work.

However, the sending of requests means that each update will accumulate at least twice as much network latency as updates in the server push system. In the rest of this paper, we will describe how we modify the VNC system to retain the advantages of a client-pull system, while getting rid of the problems caused by this additional latency.

OUR IMPROVEMENTS TO VNC

Our changes to VNC take the form of adding a *Message Accelerator* proxy between the client and server. We run the Message Accelerator on the same machine as the server, but it could also run on a separate machine close to the server (i.e., to minimize delay due to network latency). The Message Accelerator requests updates from the server, and forwards them to the client. It is invisible to both client and server. The server believes it is communicating with a client,

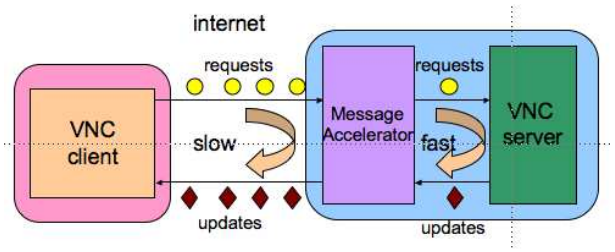

Figure 2. The Message Accelerator quickly acquires an update from the server, then pipelines sending the update to the client.

and the client with a server.

Since the Message Accelerator is on the same machine as the server and communication does not have to travel over the network, it can make requests in a client-pull manner at a much faster speed than the actual client. It requests new updates at the rate that it can send them to the client. Since the client is single-threaded and performs reads interspersed with processing, it will never read at a faster rate than it can process. The Message Accelerator checks to make sure it can write to the client's socket without blocking before it sends the update, so it will never send at a rate faster than the network can handle. As a result of all of these factors, the Message Accelerator sends updates at the client's natural processing rate, regardless of network latency.

Goals

The goal of our improvements to VNC is to create better user-perceived performance, focusing on video communication in the case of high latency between the server and client. We cannot change the time between when an action (such as a mouse movement or keystroke) is taken on the client machine and when the client receives an update that reflects that action. What we can change is the number of updates the client receives between these two events. The more updates the client receives, the less jumpiness the user will experience. This is especially important for video applications, where the number of updates per second the client receives becomes the number of frames per second the user sees.

We would like our changes to be as "plug and play" as possible, involving little to no effort on the part of the user. With that in mind, we have designed our system so that our improvements do not require changing any of the existing client or server code. In addition, if for some reason the Message Accelerator cannot be run on the same machine as the server, it can be run on another machine near the server in the network with only a slight increase in overhead. With these design choices, we avoid the issues of parallel code maintenance, as any VNC system that adheres to the standard RFB protocol should be compatible with our server [12].

Implementation

The Message Accelerator works by pipelining updates to the client. Since the Message Accelerator is running on the same machine as the server, it can very quickly send requests to the server and receive updates in response. The Message

Algorithm 1 The Message Accelerator Central Loop

```
while(True) {
  select();

  if (can read server) {
    updates[received] = read(server);
    received = received + 1;
    unansweredRequest = False;
  }

  if (can write client) {
    if (sent < received)) {
      write(client, updates[sent]);
      sent = sent + 1;
    }
  }

  if (can read client) {
    request = read(client);
  }

  if (can write server)   {
    if ((!unansweredRequest)
       && (received - sent < 1)) {
      write(server, request);
      unansweredRequest = True;
    }
  }
}
```

Accelerator can duplicate and resend client requests, without needing to wait for a new request from the client. It then sends the updates from the server to the client at a constant rate, regardless of network latency. This is illustrated in Figure 2.

The VNC session starts up with an initialization period during which the Message Accelerator simply forwards everything sent to it by the client to the server, and everything sent to it from the server to the client. This period is where the client sends the server information such as the user's password, what encoding the client would like to use, and other parameters. It is necessary for the Message Accelerator to simply forward this information, since there is no way for it to know these parameters ahead of time.

Once the initialization process is over, the Message Accelerator settles into a loop, illustrated in Algorithm 1. This loop constantly polls to check if either the server or client are available for reading or writing. The client sends short ten byte requests that the Message Accelerator saves and sends to the server. The server sends large updates (an average size for typical desktop video is around 0.9 MB) that the Message Accelerator forwards to the client. The Message Accelerator will often receive partial updates from the server, and will have to assemble an update over many reads.

If the Message Accelerator can read from the server, it reads everything available, and then parses it to see if it is a complete update. If the update is complete, it marks that the server is ready to receive the next request, since in a client-pull system we do not want to send a request until we have

fully received an update. If the update is not complete, it saves what it has received so far and waits for more. After it has read and parsed the available information from the server, the Message Accelerator updates variables that keep track of the number of complete updates and bytes of partial updates it has received.

If the Message Accelerator can write to the client, it writes as much as it can, and then records how much it has written. When the Message Accelerator can read from the client, it reads a request, and saves it to send to the server. Since the client always sends a single complete request, the Message Accelerator does not have to do any sort of book-keeping for it. If the client's display dimensions change, the client will change the rectangle dimensions in the request it sends, and all the updates the server sends after it receives this request will reflect that change.

When the server is available for writing, the Message Accelerator first checks to make sure that it has received a complete update since the last time it sent a request. It then compares the number of updates it has received from the server to the number of updates it has forwarded to the client. If it has sent all of the updates it has received, it sends a request to the server. Otherwise, it does nothing.

The Message Accelerator sends to the client whenever the client will not block its write, similar to the send mechanism in the THINC system [2]. Because we check to make sure that the write will not block before we send it, and the client is single-threaded and intersperses its read calls with its processing of the update, the rate at which the Message Accelerator can send to the client is controlled by how fast the client can process the updates. However, because the Message Accelerator sends requests to the server even if it has not yet received a new request from the client, it is not affected by network latency in the same way as a simple client-pull system.

The Message Accelerator sends requests to the server at the same rate that it is capable of sending updates to the client. It must keep updates buffered to make sure it always has something ready to send to the client. By buffering the bare minimum of updates, we minimize the effects of changing send rates. If the Message Accelerator was buffering a large number of updates and the speed at which it was sending to the client changed, these buffered updates would appear in fast motion (i.e., fast forward fashion) if the sending rate decreased, and in slow motion if the sending rate increased. In addition, perception of any user input would be delayed until all of the buffered updates had been sent. The maximum number of updates that the Message Accelerator should have buffered at any time in order to always have something ready to send to the client is $\lceil(\frac{serverprocessingtime}{acceleratorsendtime})\rceil$. Because it generally takes the server less time to process an update and send it to the Message Accelerator than it takes the Message Accelerator to send that update to the client, this number is typically one.

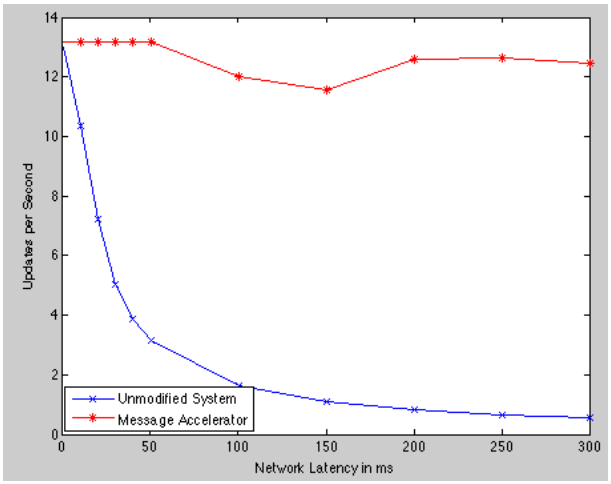

Figure 3. The median update rate rapidly decreases in the unmodified system, but stays constant with the Message Accelerator.

EXPERIMENTAL DESIGN AND RESULTS

Experimental Design

As described in Section 3.1, after initialization the VNC Client goes into a loop where it reads and processes an update, sends a request to the server, and then waits for the next update. To compare the performance of a standard (i.e., no Message Accelerator) and an improved (i.e., with Message Accelerator) system, we instrumented the client to measure how long it takes for the client to go through each iteration of this loop. Since the client does this for each update it displays, the faster it goes through this process the faster it displays updates. Since this measurement is being taken in the client, it is blind to whether or not the system is using the Message Accelerator.

To test the systems, we need a media intensive application with many framebuffer updates per second. We used an episode of the television show *Angel*, playing on a loop in the VLC media player. Our testing was done with the client running on a Dell Vostro 200 machine with the Ubuntu Intrepid Ibex OS, and the server and Message Accelerator running on a Dell Optiplex 755 with the Fedora OS. All TCP settings were at the default.

To simulate network latency, we use the *netem* Network Emulator to insert artificial network latency [9]. To generate measurements, we leave the server running, and run a script that runs the client connected directly to the server for one minute, and then runs the client connecting through the Message Accelerator for one minute. It goes through this loop ten times. All of our experiments were run using the raw encoding of both systems. By running first unmodified VNC and then VNC with the Message Accelerator, any network or processing effects outside of our system will affect both systems. All reported times are in ms (milliseconds).

Results

As shown in Figure 3 and Table 1, the unmodified VNC system is drastically impacted by network latency. With net-

Latency	Unmodified System Median	Mean	St Dev	Message Accelerator Median	Mean	St Dev
0	76.9	77.3	4.5	76.0	76.1	2.6
10	96.8	101.5	26.7	75.9	76.3	7.6
20	138.6	150.5	69.3	75.9	77.2	14.5
30	198.7	225.3	122.1	75.9	76.8	22.0
40	258.4	301.4	174.9	75.9	79.6	41.9
50	318.6	382.4	235.0	76.0	80.2	45.3
100	620.5	925.7	1117.9	83.1	111.8	172.0
150	920.6	1544.6	1243.3	86.4	128.3	187.1
200	1220.2	2313.1	1861.8	79.5	157.6	317.9
250	1520.8	3183.6	2500.7	79.2	182.4	366.5
300	1820.9	3723.3	2842.7	80.4	252.7	694.2

Table 1. Effects of network latency on client update length, in ms per update. The time per update rapidly grows in the unmodified system, but stays constant with the Message Accelerator.

work latency of just 20 ms, the client processing rate has almost halved. At network latency of 150 ms, the client processing rate is ten times lower than it was with no network latency.

In contrast, the results from VNC with the Message Accelerator show that there is no significant impact on the median update time from network latency until the latter goes beyond 50 ms. At 150 ms of network latency, the Message Accelerator system is outperforming the unmodified system by an order of magnitude.

Figure 3 displays the median rate of both systems in updates per second. The dip in rate that occurs around 150 ms of network latency with the Message Accelerator was a phenomena that occurred consistently in testing. Preliminary experiments indicated that altering the TCP window size moved or eliminated this drop in rate, so at this time we believe it is an artifact of our default TCP window size.

In the results from both the systems, we see that the mean is significantly higher than the median. This is due to extreme outliers, where the update takes significantly longer than normal. Some of these outliers are the first update sent in a session. The first update takes a long time to send because things are not yet being buffered, and is much larger than the later updates because it contains the entirety of the framebuffer. There are also outliers due to packets being lost or sent out of order, since packets resent by the TCP protocol are affected by the network latency. These outliers also result in the high standard deviation.

On a qualitative and subjective note, while watching the videos, the unmodified system becomes noticeably jittery with just 20 ms of latency, and becomes completely unwatchable with 100 ms of latency. At 100 ms, the unmodified VNC system demonstrates user observable delay while rendering single frames, and the video becomes a series of semi-related frames with no illusion of movement between them. In the VNC system with the addition of our Message Accelerator, the video is still very watchable even at 300 ms, with very little noticeable lag or jitter.

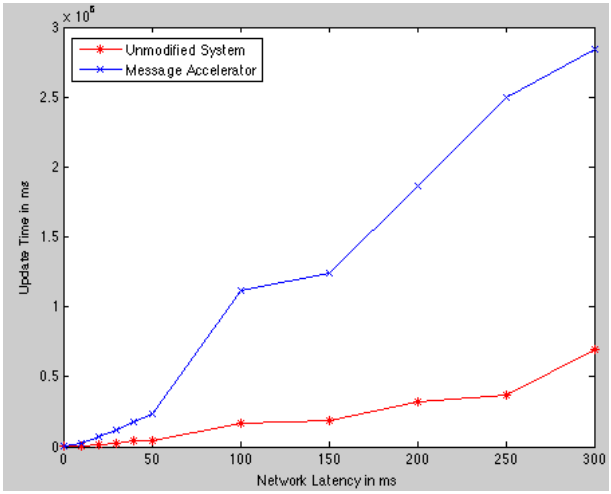

Figure 4. Standard deviation of client update times in ms. The standard deviation quickly grows in the unmodified system, reflecting the uneven rate at which frames of video are displayed.

As shown in Figure 4, we also observe that the standard deviation rises much more sharply in the unmodified system, quickly becoming ten times higher than that of the system with the Message Accelerator. This means that difference in the time it takes different frames to render on the client is much larger in the unmodified system. Disparate rendering times such as this mean that video plays at a jerky, uneven rate, alternately appearing sped up and slowed down.

Overall, we see that adding the Message Accelerator to VNC drastically improves the frame rate under network latency, even when the latency is not that large. With the Message Accelerator, the time it takes an update to be displayed remains consistent over added network latency, even when the latency becomes quite high. By both qualitative and quantitative measures, the system is much improved by the addition of the Message Accelerator.

THE EVOLUTION OF THE MESSAGE ACCELERATOR

Over the course of designing our system, we made many changes from our original design. Our system evolved as we

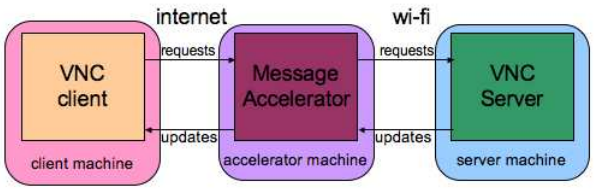

Figure 5. In our original Design, the message accelerometer was on a separate machine

learned more about the intricacies both of VNC and TCP. Our first design was based purely on the theory of how VNC should work, while our final design reflects the details of its implementation. As in many cases where theory meets design, the reality of how the system worked meant completely redesigning elements of our program. But once we started working with the program as it actually ran, rather than our ideas of how it should run, we found that it offered intuitive ways for our Message Accelerator to work with it, and that we could use aspects of the program that we had not initially known existed to achieve much faster performance when combined with our Message Accelerator. We offer this section on the evolution of our Message Accelerator to provide some insight on the inner workings of VNC and the lessons we learned.

The Original System

In our original design for the Message Accelerator, it would have resided on its own machine, one that was, approximately one wifi hop away from the client machine. In this position, the latency between the Message Accelerator and client would be small, e.g., around 3 ms in a typical wireless-access network environment. With its ability to quickly communicate with the client, the Message Accelerator would measure the rate at which the client was capable of processing updates, and then send requests to the server at this rate. Because the Message Accelerator was not waiting to receive a new update before it sent the next request, it could send requests and receive updates at the rate at which the client was processing them, regardless of network latency. The Message Accelerator would then buffer the updates from the server, and dispense them to the client in a client-pull fashion, sending a complete update in response to a request. Our original design is illustrated in Figure 5.

The Message Accelerator without Update Delimitation

One of the first problems we discovered with this system was that waiting until the Message Accelerator had received the complete update from the server before we sent it to the client was causing a huge performance hit. The Message Accelerator was reading the update in from the server one packet at a time, and waiting until the Message Accelerator had gotten a complete update before sending it to the client, then sending it to the client, which then read it in a little bit at a time, was causing a huge amount of overhead. With no network latency it was taking an average of 103 ms for the client to receive and process an update with the Message Accelerator, as compared to 77 ms without the Message Accelerator. We changed our Message Accelerator design so that instead of collecting a complete update and sending it in response to a client request, the Message Accelerator forwarded everything it received from the server to the client immediately. This meant that the client could immediately process as much of the update as it had received, taking advantage of its interleaving of network reads and update processing.

The Message Accelerator no longer had to parse the updates as it read them in, or have any internal representation of an update. It could simply read data in from the server and immediately forward it to the client. This also meant that our system was no longer a true client-pull system, and no longer had the built-in client-pull safeguards. To avoid overloading the client, we were relying on a combination of sending requests at the same rate the client was processing, and checking to make sure writes to the server would not block. This change significantly increased performance, bringing the time it took the client per update down to an average of 87 ms.

The Server Accelerator and Client Accelerator

We next discovered that sending requests to the server in a non-client pull fashion was causing delays. The server is designed so that if it receives multiple requests before it is ready with an update, it will just send one update in response. When the Message Accelerator sent requests at a very fast rate, the server would frequently process and then ignore requests that came too close together, so it was doing all of the work of reading and parsing a request without sending more frequent updates in response. We expected there would be a performance advantage in sending requests to the server in a client-pull fashion, but with the Message Accelerator on a machine next to the client, we could not do that without suffering from network latency in the exact same manner as a system without the Message Accelerator.

So we added another accelerator, one that was placed on the same machine as the server. This new *Server Accelerator* read requests from the *Client Accelerator*, and forwarded them to the server in a client-pull system, then forwarded the updates it got from the server back to the Client Accelerator. When the Server Accelerator was able to write to the server, it checked if it had received a new request from the client, and if it had been sent a full update in response to the last sent request. If both those checks passed, it sent a new request to the server. We were still depending on the Client Accelerator to set the rate at which the requests were sent.

Automatically Detecting Request Rate

Our original design for the Message Accelerator had it automatically determining the amount of time it took the client to process the request, completely separate from the time it took the update to get to the client over the network. The Message Accelerator would then send requests to the server at that rate. In this way, the system would be able to run at optimal speed, regardless of network latency. If the rate at which the client received requests was reflected in the measurement of how fast it was processing, we could suffer from a drift effect in the feedback loop, where the rate

at which the Message Accelerator sent requests got slower and slower in response to the client receiving and processing messages slower and slower, which in turn was due to the slowing Message Accelerator request rate.

Exactly how to measure pure processing was unclear, especially given the interleaving of network reads and processing. We once more had to add knowledge and processing of the updates into the Client Accelerator, after taking them out when the Client Accelerator was simply forwarding all data from the server. However, the Client Accelerator continued to forward data from the server immediately after receiving it, sending partial updates to the client, in order to keep the performance advantages.

Measuring the time from when the Client Accelerator began sending the update to when it received the next request from the client would completely capture the time it took the client to process the update. However, it would also be highly affected by the time it took to send the update to the client, especially now that the Client Accelerator was forwarding partial updates as it received them. We tried measuring the time from after the Client Accelerator had finished sending the update to when it received a new request from the client, but that returned very small times that did not capture the amount of time the accelerator was spending processing, again due to the interleaved reads.

In experiments with the Server Accelerator, we had observed that the server sent updates to the Server Accelerator much faster than the Server Accelerator could send them to the client. This resulted in a slow-motion effect when the client displayed the updates, since they were being shown on the client at a much slower rate than they had been generated on the server. To limit this effect, we put in checks so that the Server Accelerator would only prefetch a limited number of updates, and would wait to send requests to the server until it had sent these updates to the client. We noticed that with higher network latencies, the Server Accelerator was sending requests to the server at the rate at which it could send updates to the client, since this was higher than the rate at which the Client Accelerator was sending requests. Because we were checking that the client would not block before we wrote to it, the Server Accelerator could not send faster than the network could handle. Because of the interleaving of network read and processing, the Server Accelerator could not send faster than the client could process. Since the Server Accelerator is always sending as much data as it can without blocking regardless of how long it takes to hear back from the client, it is not affected by network latency in the same way as a purely client-pull system.

We removed the Client Accelerator, and had the Server Accelerator send to the client as fast as it could without blocking. When the Server Accelerator can write to the server, it checks to see if it has received a full update in response to the previous request, and if it has already sent all of the updates its received to the client. If both checks pass, it sends a request to the server.

With this simple mechanism, we no longer had to worry about separating processing time from network time. We also no longer had to calculate an explicit rate at which to send updates. By limiting the number of updates the Message Accelerator prefetched, we ensured that the updates were being generated by the server at the same rate they were being received and processed by the client, and thus we avoided any slow motion effects.

TCP Window Size

In all versions of our accelerator that featured the Client Accelerator, we found that making changes to TCP window size had a significant effect on performance. We experimented with very small and very large window sizes, and discovered that the optimal performance was when window sizes were large enough to allow a number of updates within a TCP window, but small enough that dropped packets were discovered relatively quickly. The TCP defaults resulted in our system running with a median time of 257 ms per update with a 100 ms network delay. Changing the TCP window size to 32 MB resulted in a median time of 96 ms with a 100 ms network delay, and additionally setting the Selective Acknowledgment flag got our median time down to 85 ms.

However, when we removed the Client Accelerator and once more had just the server and client machine, we found that changing the TCP window size and flags had very little effect on performance. With a 100 ms network delay, the Message Accelerator had a median time of 79 ms with the default settings, a median time of 81 ms with a window size of 32 MB, and a median time of 79 ms with the higher window size and the Selective Acknowledgment flag.

FUTURE WORK

Using an adaptive message-accelerating proxy to improve the performance of traditionally server-client applications may be an approach that can be extended to many systems. The message modifying program offers a simple way to improve performance or add additional features, is easy to deploy, and works with existing binaries. It does not suffer from problems of parallel code maintenance, and will continue to work as long as the message format between the client and server remains the same. The proxy can even be deployed to a different machine from the server, and still offer performance advantages.

There are additional ways that a modifying proxy could be used with VNC. The proxy could dynamically tightly compress updates when network speeds were low, and uncompress when the client device had low batteries or other computational issues. With an additional client application, the server application could encrypt updates, and the client application could decrypt them either on the client machine, or on a machine with a trusted network connection to the client. The server application could also perform machine vision tasks such as object detection or face recognition.

Clearly, adding an adaptive proxy to a client system offers any number of ways to improve performance or transform data. Updates can be buffered, information can be cached, or

messages can be modified in a wide variety of ways. Many systems have taken these approaches, though we are not aware of any other system that uses a proxy for message acceleration.

CONCLUSION

The Message Accelerator proxy improves the performance of VNC under high latency conditions tenfold. Even with small amounts of latency, it performs as well as or better than the unmodified VNC system. It is easy to install, requiring no recompilation of client or server code. The Message Accelerator is especially valuable for video applications, where it virtually erases the effects of network latency. A video displayed using the Message Accelerator system will look the same with a 300 ms network latency as it does with a 3 ms network latency, while a video playing on an unmodified system will take ten times as long to display updates when latency is raised to 300 ms as it would at 3 ms.

While developing the Message Accelerator system, we learned many things about the operation of a VNC system, many of which surprised us. The client's interleaving of processing and network reads caused us to change our original design for the Message Accelerator, but in the end it provided us with a simple mechanism for determining the rate at which to request and send updates. The reduction in server performance when requests were not sent in a client-pull fashion caused us to move the Message Accelerator onto (or near) the server machine, which ended up gaining us performance not only due to our use of client-pull, but also by eliminating the need to read and write to a third machine on the network.

VNC with the Message Accelerator offers a way for context-aware applications with large data and computational requirements to be run on powerful stationary servers while the user travels with lightweight devices. Since the client needs only to be responsible for I/O, it can even be deconstructed into a collection of embedded devices. With the Message Accelerator, display updates can be generated on the server and sent to the client in quick, steady fashion, regardless of network latency. The user can travel freely without affecting performance quality, and context-aware applications can use as much computing power as they require to fully process the data-rich environments of every day life.

REFERENCES

1. L. Arnstein, R. Grimm, C.-Y. Hung, J. H. Kang, A. LaMarca, G. Look, S. B. Sigurdsson, J. Su, and G. Borriello. Systems support for ubiquitous computing: A case study of two implementations of labscape. In *Pervasive 2002*, pages 30–44. Springer-Verlag, 2002.
2. R. A. Baratto, J. Nieh, and L. Kim. THINC: A Remote Display Architecture for Thin-Client Computing. Computing Science Technical Report CUCS-027-04, Department of Computer Science, Columbia University, 2004.
3. T. Haraikawa, T. Sakamoto, T. Hase, T. Mizuno, and A. Togashi. μvnc over plc: a framework for gui-based remote operation of home appliances through power-line communication. *Consumer Electronics, IEEE Transactions on*, 48(4):1067–1074, Nov 2002.
4. A. Hasedawa and T. Nakajima. A user interface system for home appliances with virtual network computing. *Distributed Computing Systems Workshop*, 0:0229, 2001.
5. A. Lai and J. Niegh. Limits of Wide-Area Thin-Client Computing. *Proc. SIGMETRICS 2002*, pages 228–239, 2002.
6. C. Lawton. 'Dumb Terminals' Can Be a Smart Move. *The Wall Street Journal*, January 2007.
7. S. Lohr. For Networks, Thin is In. *The New York Times*, September 2007.
8. T. Nakajima. How to reuse exisiting interactive applications in ubiquitous computing environments? In *Proc. 2006 ACM symposium on Applied computing*, pages 1127–1133. ACM, 2006.
9. NetEm. `http://www.linux-foundation.org/en/Net:Netem`.
10. J. Nieh, S. J. Yang, and N. Novik. A Comparison of Thin-Client Computing Architectures. Computing Science Technical Report CUCS-022-00, Department of Computer Science, Columbia University, 2000.
11. J. Nieh, S. J. Yang, and N. Novik. Measuring thin-client performance using slow-motion benchmarking. *ACM Trans. Comput. Syst.*, 21(1):87–115, 2003.
12. T. Richardson. The RFB Protocol. Technical report, RealVNC Ltd, 2007.
13. T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *Internet Computing*, 2(1):33–38, 1998.
14. B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The interactive performance of slim: a stateless, thin-client architecture. In *Proc. SOSP '99*, pages 32–47. ACM Press, 1999.
15. K. Takashio, G. Soeda, and H. Tokuda. A mobile agent framework for follow-me applications in ubiquitous computing environment. In *Distributed Computing Systems Workshop, 2001*, pages 202–207, 2001.
16. Tight VNC. `http://www.tightvnc.com/`.
17. P.-L. Tsai, C.-L. Lei, and W.-Y. Wang. A remote control scheme for ubiquitous personal computing. *Networking, Sensing and Control, 2004*, 2:1020–1025 Vol.2, 2004.
18. D. Tynan. Think Thin. *InfoWorld*, July 2005.