Optimising Thin Clients for Wireless Active-media Applications †

Cumhur Aksoy and Sumi Helal

Computer & Information Science & Eng. Department
University of Florida, Gainesville, FL 32611-6120, USA
http://www.harris.cise.ufl.edu/projects/wirelessTC

helal@cise.ufl.edu

Abstract

The thin client model is uniquely suited for low-bandwidth mobile environments, where resource-poor devices may need to access critical applications over wireless networks. In this paper, we argue for the significant advantages of using thin clients, and in the same time point to how high network latency could limit its utility. We introduce the concept of localization as an optimization approach to reduce the ill effect of high latency on thin client performance, especially in active media applications. We present our localization approach and algorithm, and show the results of four experimental case studies that quantify the benefit of localization, and the potential success of using thin clients in high latency wireless networks.

1. Introduction

With the proliferation of mobile devices and wireless networks, mobile computing is becoming a reality, touching everyday life through palm computers, cell phones, laptops, and other fancy gadgets. Today, there exist two types of networks and two types of computing platforms. Fixed and mobile devices have to communicate through fixed and/or wireless networks in a blend that is seamless to the user [1]. Moreover, the characteristics of networks and devices can vary significantly. The limitations in this new environment are either due to the nature of the resource-poor computing devices, or the low bandwidth, high latency, unreliable network connections.

The classical client/server model, which relies on reliable networks and always-available clients can not adapt to the nature of this new environment. Several extensions have been introduced to the client/server model (surveyed in [6]), ranging from the proxy approach (adding a proxy between the client and the server, on either side or both), to adding disconnected operation capabilities to client/server applications (e.g. the Coda file system [5]), to using mobile agents as an adaptive computing model and a disconnection-friendly transport. However, most of these extensions are either application-specific, requiring modifications to pre-existing applications, or fall short of offering end-to-end guarantees.

Adopting the thin client model into the mobile and wireless domain has received cursory consideration [6,7], but hasn't been investigated deeply enough. The thin client model has the following advantages, which make it a competing model to the proxy-based approach:

- The thin client is relatively a small piece of software and therefore can fit easily in resource poor portable computers, especially hand-held and Palm computers.
- The thin client is all that needs to be installed in the mobile device to provide access to all applications available on the fixed network server. This is a significant advantage given the limited resources (including storage and operating system) of the mobile device. In a way, the thin client can be envisioned as a mobility-support infrastructure, where applications are made mobile by placing them on the fixed network server that caters to the wireless thin clients. A significant consequence of this is that a compute-intensive application such as mathematical modeling that may require a supercomputer could very easily be ported into a wireless Palm computer or a hand-held PC (and of course laptops). The effect is equivalent to

† This research was funded by a generous grant from Citrix System Corporation (Grant No. 4514227-12)

bringing the supercomputer out and close to the hands of the user (a realization of a *pocket super computer*!)

- The thin client approach provides a crucial system value, which is scalability. Currently, state of the art proxies for mobile users can perform filtering and data transformations. But these techniques are not scalable in several applications. For example, very little can be done to a simple text editor application. Imagine editing the tail of a 5MB file. Using proxies, the entire file needs to be downloaded to the mobile computer, which could take several minutes under a slow wireless network. Using a thin client, the text editor is invoked and the file is browsed on a fixed network server, and only the block of data near the end of the file is downloaded (as display frames) to the mobile computer. Therefore, the thin client approach provides higher scalability for applications that deal with large data sets.
- Thin clients eliminate management needs on the mobile platform and therefore provide cost-effective application extensibility.
- Thin clients provide high reliability because the applications and data are stored and maintained on a high reliability server in the fixed network at all times.

However suitable thin clients are in the mobile environment, they do not promise a perfect solution. The thin client model, which requires short yet frequent message exchanges, performs poorly in high latency networks because the application's response time decreases along with latency. With the advent of third generation wireless technology, the bandwidth is expected to soar. However, high latency will be inherited and is expected to linger for a longer time.

In this paper we investigate optimizations to overcome the limitations of the thin client model in high-latency mobile networks. The work is focused on decreasing the rounds of communication between the thin client and the server. Our optimization approach is application-independent. Specifically, our approach is to *localize* the active parts of an application (active components), which exhibit a behavior of repeating display structures.

Section 2 will explain the concept of localization, and will give a taxonomy of localization approaches. Section 3 will describe the Ultra Thin Client Prototype that we have developed and used to implement the proposed optimizations. Section 4 details localization algorithms of active components. Experimental results and improvement measurements of the proposed optimizations are given in section 5. Related work is covered in section 6, and finally, conclusion and future work are given in section 7.

2. Localization

Localization, in a thin client sense, is a return to the features of the classical client/server model. It is a "fattening" of the thin client by assigning additional responsibilities to it. The degree of localization can vary from none (pure thin client) to having a local copy of the application (full client/server). The following figure depicts the degrees of localization that could be required depending on the connection mode.

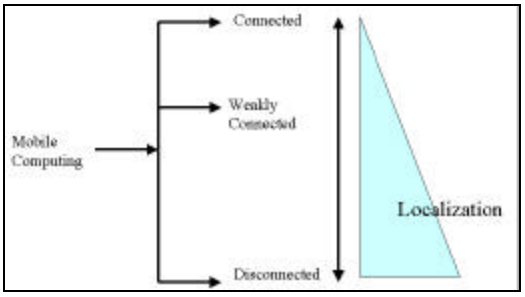

Figure 1. Localization levels with respect to connection quality

In the fully connected mode, there is no need for localization. However, as the connection quality decreases, localization has to increase to take over part of the tasks of the server. Figure 2 shows a categorization of the possible entities that could be localized into the mobile computer.

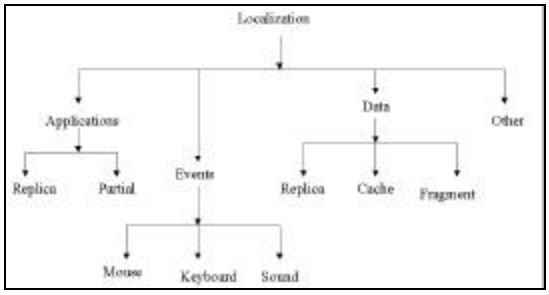

Figure 2. Possible localization optimizations

Application localization can be achieved by pre-installing the application in the mobile client. This would reduce the communication between the thin client and the server, requiring only data to be transferred. Depending on the quality of the connection the thin client can decide to switch from the thin client based application to the local replica, or vice versa. The switching operation would need

to transfer all the data and application resources to the thin client or, update the server application's state in switching to the thin client mode. Hoarding all the data to the thin client, or using a proxy or fragmenting the data also represent the possible localization options. Other possible localization schemes can be based on events such as keyboard, mouse, and voice inputs. A keyboard event can be localized by handling the keyboard input locally at the thin client without transferring them to the server. In the same sense, mouse movement and mouse pointer location can be processed locally whereas mouse clicking, scrolling or crossing over an active screen area (such as sensitive maps) is processed remotely.

Localization can also be categorized as being *application transparent* or *application specific*. In the first case, the system treats the application as a black box assuming no help of any kind while localizing parts of the application. This type of localization works with any type of application regardless of its specific characteristics. Application specific localization, on the other hand, requires the application to present interfaces and information to the thin client system, so that the system might decide on localization policies suitable for that application. In a web browsing application, for example, the web server can send the HTML page to the thin client for analysis and to make localization decision (e.g., pull animated GIFs for local play if player is available). Application specific localization requires applications to be designed and developed with thin client support, which means current applications have to be rewritten or modified. Another problem is that, it would be difficult to achieve a universal way to describe thin client interfaces.

This paper focuses on application-transparent localization. Therefore, the proposed optimizations apply to any kind of application running in a thin client environment.

3. The Ultra Thin Client Prototype

A very basic thin client system is implemented from scratch, which gave the capability to test the ideas and quantify the results. The Ultra Thin Client Prototype (UTCP) is composed of two main modules: the server and the client. Figure 3 gives an illustration of UTCP.

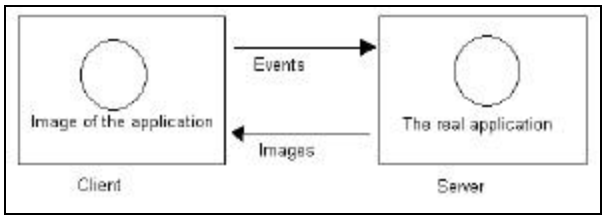

Figure 3. The Ultra Thin Client Prototype (UTCP) system architecture

The UTCP prototype is a single client, single-window, and single-application system. So it doesn't support multiple clients logging on the server and running several applications at the same time.

The server is mainly responsible for managing the real copy of the application, sending display updates to the client and tunneling the events received from the client to the application. The server samples the application's window every 400-500 milliseconds, and checks if the display image has been changed. If so, it compresses the difference image and sends it to the client. Only difference images (the images obtained by subtracting the new image from the previous sampling) are being transferred. At the client, the difference image, is pasted onto the previous image, giving the new display of the application.

Sending the difference image has advantages because typically a small part of the display changes, whereas a complete screen change is infrequent. Moreover, difference images give better results when being compressed prior to the transfer. Run length encoding was employed as the method of compression. The algorithm for the server can be summarized as follows:

```
PreviousApplciationDisplay =
                    (Array of zeros)
Every N milliseconds {
  CurrentDisplay = Get Curr. Display;
  DiffImage = CurrentDisplay -
         PreviousApplciationDisplay;
  If (DiffImage <> All zeros) {
       Compress DiffImage using
              run-length encoding;
       Check for optimizations;
       Transfer the compressed
              bitmap to the client;
  }
  PreviousApplciationDisplay =
                     CurrentDisplay;
}
```

Optimizations on this algorithm will be discussed in the next section. The client, on the other hand, manages the image of the application; it takes the display from the server and pastes it on the screen. The client manager also keeps track of user input and channels them to the server if necessary.

The system is developed under Windows NT 4.0 Service Pack 4, using Visual Basic 6.0 Enterprise Edition, Service Pack 3. Visual Basic provided rapid development as well as a nifty user interface and graphical output. Win32 API was used in operations where performance is critical (such as taking the difference of images).

UTCP server is run on a Pentium II- 266 computer with 256MB of RAM and a screen resolution of 1024x768 high-color, where the client is a Pentium II - 233 laptop with 96MB of RAM and 800x600 high-color screen resolution. The laptop has a wireless network card that supports 2Mbits/s data transmission rate.

4. Localizing Active Components

4.1. Active Components

Active components are parts of the application, which present a recurring sequence of images, sound or data. A perfect example is an animated GIF, where several images are being displayed repeatedly. Animated GIFs are used in this paper to demonstrate the optimizations because they provide an easy way to evaluate performance and accuracy of the algorithms. The same ideas can apply to voice, or data that represent similar characteristics.

An animated GIF composed of n-images can be represented as a state machine where each state is an image and state transitions occur in a timely manner with respect to the GIF's timing information. A blinking cursor or text can also be viewed as a 2-state animated GIF. The caret or text is being drawn in the first state and erased in the second.

Depending on the sampling rate, an animated GIF of n states can be represented as n 1 or 2 state GIFs. Consider an animated GIF of 4 states with the timing in Table 1.

Table 1. States for a 4-state animated GIF

State 1	200ms
State 2	200ms
State 3	150ms
State 4	300ms

It can be viewed as a four single state animated GIFs, where each image is displayed every 850 milliseconds. On the other hand while trying to simulate an animated GIF in such a way, every state has to contain the negative (i.e. complement) of the previous state to delete it from the display.

4.2. Loop Detection

The initial optimization offerred is to detect whether the whole application display represents a repeating structure. That is, the whole display keeps looping over several images. This might be the case when a user is browsing the web and is idle while browsing a page with active components. This case incurs costs for the thin client because the same state of displays are being sent to the client over and over.

The server holds a cyclic buffer of a given size, where it stores all the difference images that are being transferred to the client. Each time the applciation window is sampled, it is added to the buffer and every image is given an image ID. A new image is compared to all of the images in the buffer to find out if it has been added earlier. Only new images are assigned new IDs.

For an application whose display is just a two state animated GIF, the buffer will have the following structure in Table 2.

Table 2. Buffer for a 2-state animated GIF

BUFFER ID	IMAGE ID
1	1
2	2
3	1
4	2
5	1
6	2

A loop detection algorithm is then run on the Image IDs to find out recurring structures. Once a loop is detected (at sampling 4 in that case), the server informs the client that it has found a loop of n states (2 in this case) and the client sets aside a buffer of n (2) images. Then the client waits for n images, buffering and also recording their arrival times. When the client fills its buffer, it goes into zero communication mode, where it displays the images in the buffer one after another, according to the timing information.

There are also situations where the display structure represents a non-deterministic loop. Then the server simply informs the client of the buffer size, and the client allocates a buffer and fills it. This time, the server just sends image IDs to the client instead of the images themselves.

After the server detects the loop, it goes into a watch mode, where it constantly samples the application display as usual and makes sure that the application obeys the loop it has detected. As soon as an image out of the loop arrives, the server passes it to the client putting it back into the dummy thin client mode.

4.3. State Explosion

When the application has more than one active component, an increase in the number of states is observed due to the different frequencies of those active components. If a browser were viewing a web page with two active components, the number of total states to capture the entire display states would depend on factors such as the number of states in each component and the frequency of those states. The number of states combined could therefore exceed the Cartesian product of the states.

Take for instance the 2-state animated GIFs in Figure 4. The first, Altavista, has a frequency of t_1 and the Sony Card GIF has a frequency of t_2.

Figure 4. Two 2-state animated GIFs overlaid

According to the above sampling rate, the states in Table 3 are formed. ACm_n stands for Active component m's n^{th} state. And "'" stands for the negative (complement) of that image.

Table 3. States for GIFs in Figure 4

STATE	TIME	DISPLAY
1	z_1	$AC1_1$
2	z_2	$AC2_1$
3	z_3	$AC1_2 + AC2_1'$
4	z_4	Nothing
5	z_5	$AC1_1$
6	z_6	$AC1_1' + AC1_2 + AC2_2$
7	z_7	Nothing
8	z_8	$AC1_2' + AC1_1$
9	z_9	$AC2_1$
10	z_{10}	$AC1_2 + AC1_2'$

Therefore, two 2-state animated GIFs usually can not be represented as 4 states but more (6 in the above case). In fact, in the experiments done with the two animated GIFs above, the server detected 16 states, 9 of which were distinct.

4.4. Active Component Extraction

For an application display with 3 active components, a buffer of 100 images was not enough for the server to detect the loops. So we tried to extract each active component from the application display and transfer and simulate them separately at the client. So two 2-state animated GIFs would be detected separately and only 4 states and timing information has to be transferred instead of 16 states.

Since the approach we are using is an application-independent one, we need to extract active components from the display of the application through image processing. The extraction process is done by using vertical and horizontal scan lines passed over each image sampled. The application seen in Figure 5 has 3 active components. Altavista is a 2-state active component with 2000 ms timing on each state. Sony Card GIF also has 2 states with 2500 ms-display frequency. The envelopes GIF has 6 states where each state is displayed for 150,150 150, 1500, 150 and 1500 milliseconds.

Figure 5. Sample application

Scan lines passed through an image in the buffer gives the following result in Figure 6. However, the active components' position in the image might produce additional active component areas, so further processing is done on Figure 6 and "smallest enclosing rectangles" - Figure 7 - are discovered. The rectangles are stored in the image buffer by storing their top-left corner and bottom-right corner.

The active component extraction algorithm is then run on the buffer. For each enclosing rectangle, the algorithm, passes over the images that have the same enclosing rectangle and enumerates the regions, discovering the number of distinct images enclosed in that area.

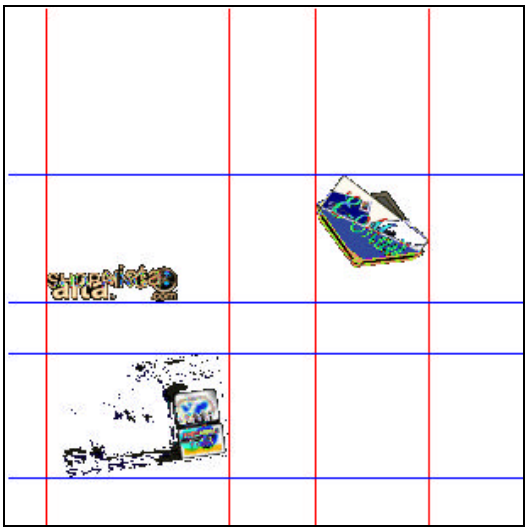

Figure 6. Scan lines

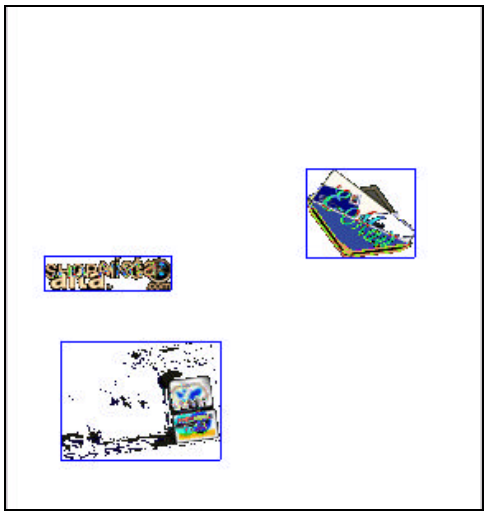

Figure 7. Optimized scan-lines

The output of the algorithm for the application in Fig. 5 is shown in Table 4. Area 32,129,29,101 corresponds to the Sony Card GIF, whereas 22,99,132,153 corresponds to the Altavista's. Other areas belong to the envelope GIF. The algorithm was run on a buffer of 100 images.

The timing information, on the other hand, is extracted from the time stamps of the samplings. The frequency of a state is approximated by calculating the average of its display frequencies. The algorithm generally performs well with error rates below 1%.

Table 4. Algorithm Output for GIFs in Fig. 5

REGIONS		IMAGES & BITMAP IDS
22,99,132,153	Imgs	0,4,6,8,11,14,17,19,22,25,28,31,33 ,35,38,41,44,47,49,51, 54,57,59,63,65, 68,70,72,75, 78
	ids	1,2,1,2,1,2,1,2,1,2,1,2,1,2,1, 2,1,2,1,2,1,2,1,2,1,2,1,2,1,2
169,247,129,206	Imgs	5,8,13,18,21,23,27,29,33,37, 42,46,48,50,52,55,57,59,62, 65,69,73,77,79
	ids	1,1,1,1,2,3,2,1,1,1,1,2,1,2,1, 2,3,2,1,1,1,3,2,3
32,129,29,101	Imgs	3,6,9,12,16,19,23,27,30,33, 36,39,43,47,50,53,56,59,64, 66,69,72,76,80
	ids	1,2,1,2,1,2,1,2,1,2,1,2,1,2,1, 2,1,2,1,2,1,2,1,2
187,231,142,182	Imgs	24,58,74,80
	ids	1,1,1,1
169,244,129,196	Imgs	1,10,34,67,71
	ids	1,1,1,1,1
183,247,156,206	Imgs	2,11,35,64,68,72
	ids	1,1,1,1,1,1
169,245,129,189	Imgs	6,15,31,39
	ids	1,1,1,1
181,247,152,206	Imgs	20,26,45,49,54
	ids	1,1,1,1,1
187,232,142,182	Imgs	24,58,74,80
	ids	1,1,1,1

For the Altavista image the first state's average display time is calculated as 2004.3 milliseconds whereas it is 2002 milliseconds for state 2. So the error rate for state 1 is 0.22% and 0.1% for state 2. However with increasing number of active components, the algorithm might produce error rates as large as 10%.

The following section demonstrates the results of our optimizations and the performance of AC extraction within the UTCP thin client prototype.

5. Experimental Results

5.1. Test Case 1

Application: Browser with one 2-state animated GIF.

Sampling Rate: 500 ms.

Active Component: Animated GIF with 2 states. Each state displayed for 2 seconds.

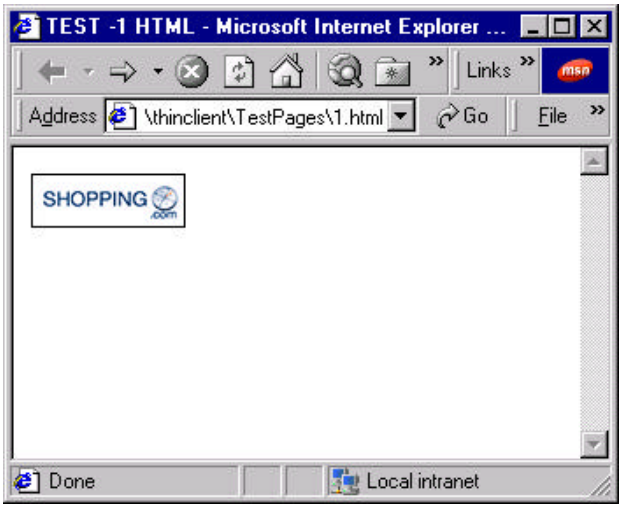

Figure 8. Test Case 1 sample screen shot

The application is first run with the thin client prototype, without performing any loop detection or localization. In this case, the total number of bytes transferred increases as time passes. The initial transfer is the largest (about 75KB) whereas subsequent transfers are around 6K since they are only difference images.

With the loop detection algorithm, 5 samplings after the initial transfer, the loop of 2 states is detected and the client is put into localization mode immediately. After that time, there is no communication between the client and the server (unless the application's loop is distracted). The total number of bytes transferred to the client in the two cases is compared in Figure 9.

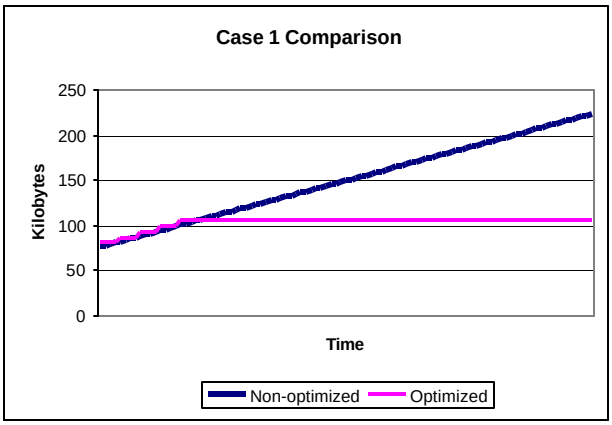

Figure 9. Cumulative bytes transferred in case 1

5.2. Test Case 2

Application: Browser with two 2-state animated GIFs.

Sampling Rate: 500 ms.

Active Component 1: Animated GIF with 2 states. States displayed for 2 seconds.

Active Component 2: Animated GIF with 2 states. States displayed for 2.5 seconds.

In the pure thin client mode without any optimizations, the application sends about 5 Kilobytes (5.01 KB) on the average on every sampling which takes place every ~500 ms.

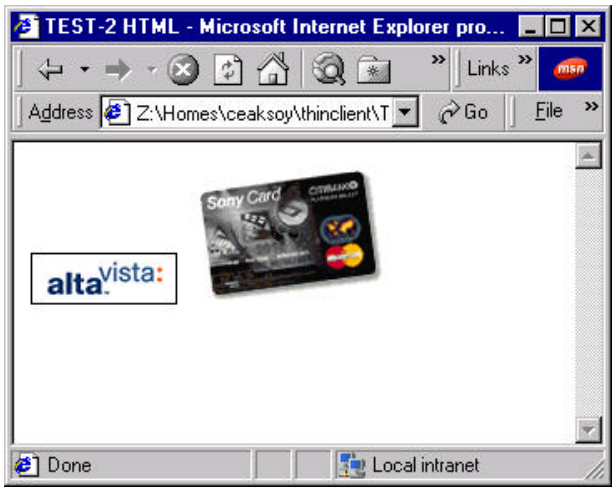

Figure 10. Test Case 2 sample screen shot

When the application is run with loop detection, the server, at the 79th sampling determines that the display

forms a recurring structure of 16 states, with 6 distinct states. It then passes the 6 distinct states to the client and puts client into the localization mode. After the 99th sampling (until all the states are transferred to the client), there is no communication between the client and the server. The amount of bytes transferred is 422 KBs.

However, with active component extraction, after the 12th sampling, the server extracts the 2 active components separately. The first AC with a frequency estimation of 2003 milliseconds (0.15% error) and the second AC with 2379 ms (4.84% error). If the buffer size were kept larger, the error rates would be smaller (around 1%). And the total number of bytes transferred is 179 KB in that optimized version. The comparison of bytes transferred is given in the chart below (Figure 11)

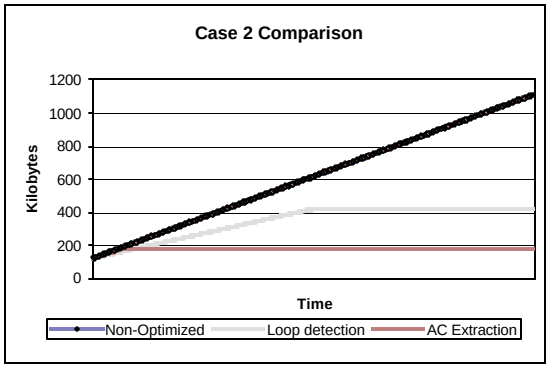

Figure 11. Cumulative bytes transferred in case 2

5.3. Test Case 3

Application: Browser with a 6-state animated GIF.

Sampling Rate: 400 ms.

Active Component: Animated GIF with 6 states. States displayed for variable amount of times and some are smaller than the sampling rate.

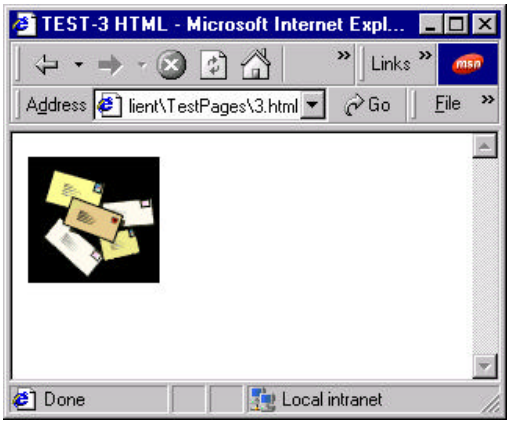

Figure 12. Test Case 3 sample screen shot

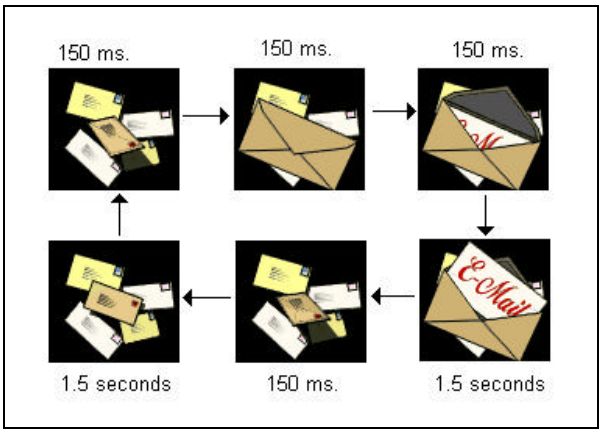

Figure 13. Case 3 seconds animated GIF state diagram

In this case, the initial transfer is 75KB and then an average of 2.5 KB (2446 bytes) is transferred at every sampling, which is done every 400 ms.

The important thing about this active component is that some of its states are displayed for a shorter amount of time than the thin client prototype can sample them, which leads to *state misses*. The server misses some of the states that are being displayed for 150 milliseconds because it is limited by 400 millisecond sampling rate. So the thin client's view of the application is not perfectly identical to the server's. The states that are missed depend on their display quantum and the starting time of the animated GIF and the sampling process. For more than 10 experiments conducted with the sampling rate of 400 ms, the server was able to catch 3 to 5 states out of 6. So the server, depending on the relative timing with respect to the active component, either misses a single state, 2 states or at most 3 states. The time the client is being started plays an important role. The loop detection algorithm can be modified to adjust its own timer to be able to capture as much states as possible.

For a perfect system, the prototype can be implemented in a multithreaded way, where by interleaving the operations, a better sampling rate can be achieved. However, there will always be a limit on the sampling rate unless the thin client is either implemented at the operating system or graphics display device driver level.

The loop detection algorithm was able to detect a loop of three states (states 3,4 and 6) after the 89th sampling. Then the client started to simulate the active component as a 3 state animated GIF. The total amount of bytes transferred is 263 KB.

With the introduction of active component extraction, the server detects 5 different active components missing a single state. The estimated frequencies are given in the Table 5. The average error rate is an acceptable 0.23%.

Table 5. Estimated frequencies for Case 3

STATE	FREQUENCY ESTIMATE
1	3607
2	Missed state
3	3606
4	3615
5	3608
6	3606
Average	3608.4
Error	0.23 %

The following chart gives a comparison of the number of bytes transferred in each experiment.

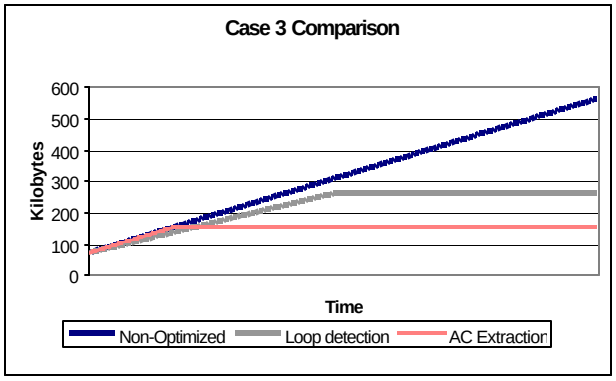

Figure 14. Cumulative bytes transferred in test case 3

5.4. Test Case 4

Application: Browser with multiple active components.

Sampling Rate: 400 ms.

Active Components: One from each test case above.

This case demonstrates a sample example where the regular optimization cannot find any loop structures between the screen shots although a buffer of 200 images was used. So loop detection is useless in this case.

Figure 15. Test Case 4 sample screen shot

On the other hand, turning on the active component extraction, the algorithm, after 50 samplings, returns satisfying results. It detects 5 separate active components. Two for 2-state animated GIFs on the left-hand side and 3 for the 6-state "e-mail" animated GIF. So it misses three states of the multi-state GIF. The following table gives the frequency approximations of the algorithm:

Table 6. Output of active component extraction for test case 4

ACTIVE COMP	ANIMATED GIF	EST. FREQ.	ERROR RATE
1	*"Alta vista"*	1960	2 %
2	*"Sony Card"*	2463	1.48%
3	*"E-mail"*	3609	0.25 %
4	*"E-mail"*	3609	0.25 %
5	*"E-mail"*	3609	0.25 %

The errors on the 2-state GIF are greater than the usual error rates; however, these are still acceptable rates. The total amount of bytes transferred is 357 KB. After transferring 357 KB there is no communication between the client and the server. The results are compared graphically on Figure 16.

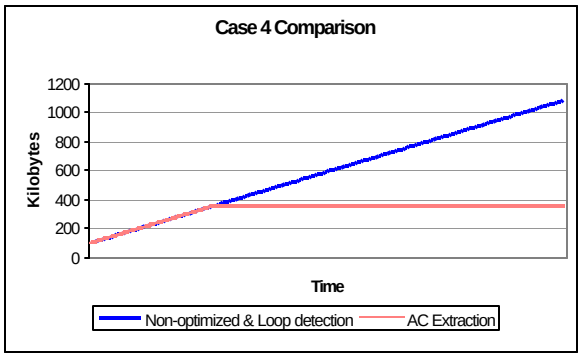

Figure 16. Cumulative bytes transferred in case 4

5.5. The Effect of Window Size

The size of the display effects the thin client's performance. The applications used above can be considered as small sized applications. Figure 17, shows how applications' display size effects the performance of the UTCP. The experiments were conducted with application sizes that allow processing time to be below 400-500 ms range. As seen from the graph the processing time exceeds 1.5 seconds for bigger display sizes.

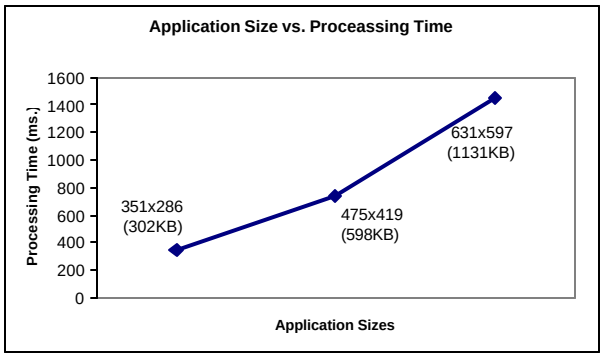

Figure 17. Application size versus processing time

We find it important to report the effect of the display size on processing time to base the reported timing results relative to the display size we chose in our experiments.

7. Conclusion and Future Work

In this paper we introduced the concept of localization in the context of the thin client computing model and its applicability to the wireless and mobile environment. We argued for the unique advantages of using the thin client in such environments and clarified how the utility of such approach is threatened by the high latency inherent in most wireless networks. We proposed application-transparent localization that allows for the active portions of an application (whatever the application may be) to be swiftly detected, isolated, and localized into the client side. We presented our localization algorithms that we have implemented within a simple thin client system that we built from scratch. We presented a series of case studies that measured the positive effect of this type of localization in terms of the number of communicated bytes between the server and the thin client.

Currently, another localization approach is underway [8], with focus on user keyboard and mouse input optimization. The research is being conducted on the CITRIX ICA thin client [2,3,4] using CITRIX' Virtual Channel SDK [9]. Unlike the Active Component localization that stems from the server side, Keyboard localization stems from the client side and provides a crucial optimization of highly interactive mobile application.

Acknowledgements

We would like to acknowledge CITRIX Corporation for supporting this research. Stephen Spector, Richard Andresen, Mike Palmer, and Jeff Russo provided helpful comments. Jörg Peters provided helpful discussion about the design of the scan line algorithms.

8. References

[1] A. Helal, B. Haskell, J. Carter, D. Woelk and M. Rusinkiewicz, "Any Time, Anywhere Computing: Mobile Computing Concepts and Technology," Kluwer Academic Publisher, September 1999.

[2] Citrix Systems Inc, http://www.citrix.com, Jan 2000

[3] Citrix' Server Based Computing – A Citrix White Paper, http://www.citrix.com

[4] Citrix MetaFrame for Windows 2000 Services Fact Sheet, http://www.citrix.com

[5] M. Ebling, M. Satyanarayanan, "On the Importance of Translucence for Mobile Computing," Proceedings of the 15th ACM Symposium on Operating Systems Principles, May 1998.

[6] Jing, J. & Helal, A., & Elmagarmid, A., "Client-Server Computing in Mobile Environments," *ACM Computing Surveys*, 1999.

[7] M. Le, S. Seshan, "Software Architecture of the InfoPad System," Proceedings of the Mobidata Workshop on Mobile Wireless Information System, Nov 1994.

[8] A. Helal and S. Ramamaurthy, "Optimizing Thin Clients for Wireless Computing via Localization of Keyboard Activities," Submitted to the International Performance, Computing, and Communication Conference, to be held in Phoenix, Arizona, April 2001. Also available as UF Technical Report number UFL-00-003.

[9] Citrix Virtual Channel Development Kit Documentation. http://www.citrix.com.