Network latency hiding in thin client systems through server-centric speculative display updating

Bert Vankeirsbilck[a], Pieter Simoens[a,b], Filip De Turck[a], Piet Demeester[a], Bart Dhoedt[a]

[a] *Ghent University - Department of Information Technology (INTEC), iMinds*
Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium
[b] *Ghent University College - Department INWE, Valentyn Vaerwyckweg 1, 9000 Gent, Belgium*

Abstract

The widespread availability of cloud computing services has revitalized interest in the thin client computing paradigm, in which application logic is executing on a remote server, typically hosted in a cloud computing infrastructure. The user interacts with a local viewer, that forwards the user events over the network to the server and accepts the returned graphical updates. An important challenge for this approach consists of the fact that at least one network round-trip time is required to present the application output that results from the user's actions. In this paper a novel speculative display update mechanism is proposed to hide the network latency from the user by speculatively updating the screen without awaiting the server response. The mechanism relies on online server side profiling of the graphical output caused by user events, based on which a finite-state model is constructed capturing the graphical behaviour of the application. Experiments with a text editor show that, once the application model is learned, speculative responses are displayed within 40 ms for over 80% of the user events, with an accuracy exceeding 70%.

Keywords: Thin Client, Speculative, Remote Display, Latency, Bandwidth Reduction, Caching

1. Introduction

The advent of cloud computing has introduced new possibilities to employ the thin client computing paradigm. This paradigm consists of a client-side viewer forwarding user events to the application logic executed on a remote server. Screen updates calculated at the server are forwarded to the client in response to the received user events. The client function is inherently limited to I/O functions, drastically reducing the requirements for computational resources. The advantages of thin client solutions are well known and include total cost of ownership reductions, simplified maintenance, data security and privacy, ubiquitous data and service access and more efficient use of resources [1, 2].

The generic architecture of traditional thin client systems is presented in Fig. 1. This traditional thin client approach implies that at least one network round trip time (RTT) is required to present the application output that results from the user's actions. More specifically, the user input must be transmitted over the network to be delivered to the application that is executed on the server, before the graphical output can be sent back for presentation by the viewer. Wide Area Networks (WAN) and mobile networks typically exhibit relatively large latencies, making the RTT the major factor influencing the quality experienced by the user. In [3] and [4], the authors quantify the users' limits for accepting bad responsiveness depending on the task they aim to fullfill. For office automation (word processing, text and figure editing, etc.), RTT values below 150 ms are shown to be acceptable, while for more interactive applications, such as gaming, 80 ms RTT is found as an upper limit for usability. In [5] it is shown that roughly half of the total round trip time is caused by packetization and propagating data over the network.

In this paper a novel speculative display mechanism is proposed to mitigate the adverse effects of network communication delay on the user experience. A Finite-State Machine (FSM) is constructed, capturing the relation between sequences of user events and resulting graphical updates. This FSM is provided to the client, such that these graphical updates can be forecasted by the client when a certain sequence of user events occurs. The validity of the speculatively displayed updates

Email address: Bert.Vankeirsbilck@intec.UGent.be (Bert Vankeirsbilck)

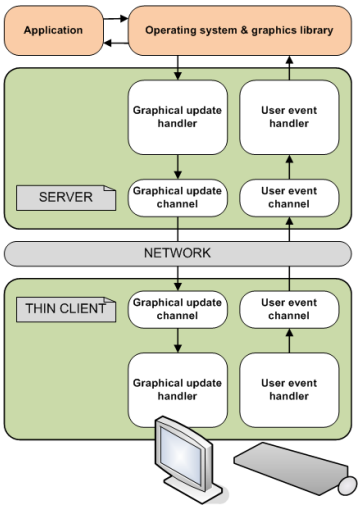


Fig. 1: Traditional thin client system architecture. The server executes the application, the viewer acts as a service-hatch, passing intercepted user actions to the server and displaying the received graphical updates.

is checked by the server, possibly resulting in corrective graphical updates and an update of the FSM.

We leverage on prior work [6], where we explored the potential of using a static image cache in thin client computing, and assessed the repetition of screen updates. Although a side-effect exists of being able to deliver the smaller graphical updates faster over the network due to higher compression achieved by difference encoding to the client cache, the main target was to decrease the bandwidth consumed by the thin client protocol. In contrast, the current paper focuses on latency mitigation, through a mechanism that involves caching and could hence additionally result in a decreased bandwidth usage.

The contributions of this paper are *(i)* a server-controlled display state profiling mechanism that mitigates network round trip times and decreases bandwidth usage, *(ii)* a theoretical basis for evaluating the applicability of the proposed mechanism and thin client protocols in general, for varying network conditions and setup configurations and *iii* the provisioning of experimental results obtained through a prototype implementation.

The paper is structured as follows: Section 2 provides an overview of related research. In Section 3, a listing of the concepts used to throughout the paper is provided. A model for the latencies involved in remote application execution is constructed in Section 4. The speculation algorithms are detailed in Section 5. The required modifications to traditional thin client architectures are explained in Section 6. Experimental validation results are presented in Section 7. Finally, conclusions and future work are described in Section 8.

2. Related work

Speculative thin client operation has been proposed in [7], restricting modification to the thin client viewer only to maintain compatibility with existing server implementations and thin client protocols. They have shown predictability of screen updates for both Virtual Network Computing (VNC) [8] and Remote Display Protocol (RDP) [9] and apply a Markov chain at viewer side to relate series of user and screen events to following screen events. However, the viewer-based approach is expected to be resource demanding due to the need for multiple image comparisons per user event. In contrast, we aim to quantify the gains obtained by relaxing the constraint of compliance to the original thin client protocols. This relieves the viewer from the image comparisons and fits naturally into the thin client paradigm where the server takes care of the heavy computing tasks. Additionally, bandwidth reduction is realized since the server can refer to graphical data already present at the viewer or can send graphical data in differential mode to increase compression.

In the context of human-computer interaction, FSMs have been used to describe applications [10]. In this field of study, FSMs are created manually by application developers to understand the operation of an application and to optimize the user interaction. Model-based Graphical User Interface (GUI) testing methodologies derive FSM models from the application source code too [11, 12], as part of automated regression tests for evaluating the correct operation of the user interface code. Automating the creation of FSM models from applications, without access to the source code is investigated in [13]. By interfacing to the graphics libraries used by the application, the GUI is decomposed into widgets. The *executable* widgets, i.e., widgets which allow user input, are filtered to supply input to. This way, *executable* widgets are traversed to reverse engineer the model of the application GUI. The speculative display mechanism proposed in our paper also relies on FSMs based on GUI widgets. We also assume no a priori knowledge of the application GUI structure, and are constrained to reverse engineering this model through monitoring user events and graphical responses. As opposed to GUI testing settings, our mechanism operates in a live environment where users interact with the application, we must resort to learning the FSM gradually as the user provides input. Some, but not all, thin client

implementations are able to intercept widget information. In our proposed speculative display mechanism, we support the creation of the application model without specifically relying on the widget library used by the application.

In [14], the authors aim to predict when graphical responses to user input will arrive at the client and put the network card into sleep mode to reduce the energy consumption of the mobile device. The authors even predict how much time is spent between the receipt of the graphical response and the next user action, to go to sleep mode between these events as well. The authors derive a FSM from the application using the source code of the application or the GUI toolkit the application is based on, and focus on obtaining statistics concerning the times spent in the states of the FSM. In our work, we have a similar concept of deriving a FSM, albeit without relying on source code or GUI toolkit, but use the generated FSM for the purpose of latency hiding. The use of a similar FSM for both speculative display operation as decision strategy to put the network card into sleep mode enables the proposed mechanism to cooperate with the cited work easily to build an energy-efficient reactive remote display system.

In [15], a server-based adaptive display pre-fetching mechanism is proposed that speculatively executes possible user events on the server and sends the related graphics to the viewer. On receipt of the user input, the matching graphical update is presented to the user. This way, using spare server computational resources, bandwidth and client memory, round trip times are effectively avoided. However, the approach requires the knowledge of multiple subsequent user actions to be applied to the application, and is expected to imply the need of reverting these actions or having parallelly executed instances of the application to guarantee the genuine application state that corresponds with the state of the client.

Our contribution focuses on reducing application level latency by learning the correlation with graphical updates result from user events. Once learned, these graphical updates are presented speculatively to the user upon receipt of the user event. To this end, we construct a FSM at the server, that consists of nodes representing graphical application states (e.g., all menu items are collapsed, a given application menu is expanded, etc.). User input triggers transitions between states, using the related graphical updates. The server-centric approach results in low complexity algorithms on the client device, which is beneficial for the energy consumption. Furthermore, the transferability of the session to other devices is ensured, as the FSM is constructed at the server which can be synchronized with a viewer upon resuming a previous session.

3. Basic concepts

This section presents the main concepts referred to throughout the paper as they are key to speculative remote display mechanisms. These are illustrated in Fig. 2 for the case of opening and closing a 'File' menu in a text editor.

3.1. Frame buffer update

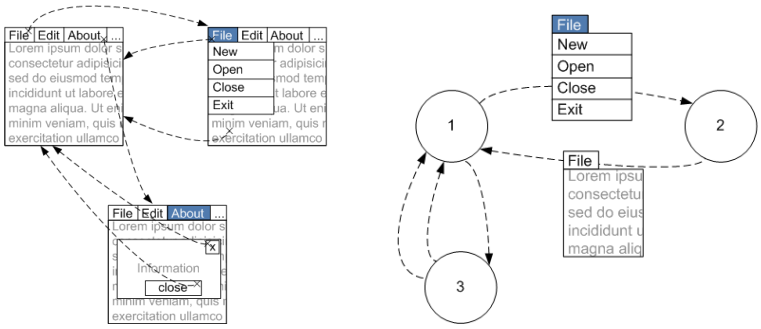
When the display must be updated, the server packages and sends these graphical changes to the viewer. These changes are called frame buffer updates (in analogy to the terminology used in VNC's Remote Frame Buffer (RFB) protocol).

3.2. Finite-State Machine

The proposed speculative display mechanism is based on a FSM, in which a state represents a specific layout of visible widgets a user interacts with. When a user interacts with these widgets, frame buffer updates bring the system to another state with a different visual layout. These frame buffer updates form the state transition between the two states as shown in the figure. Note that it is possible to have multiple transitions between states, e.g., in the case a user closes a dialog box using a dedicated close button or the 'X' button of the window decoration. Different widgets are activated, and correspondingly the graphical candy of the button press will result in slightly different frame buffer updates, but eventually, the initial state and the final state are identical.

3.3. Hotspot

The client needs to be able to link a user action to a state transition. However, there exists a many-to-one relation between user input and state transitions. For example, the user can open a menu of a graphical interface in different ways: it can be accessed by a keyboard shortcut, or any mouseclick within the region of the menu widget result in the same graphical changes on screen. In our approach, we define a hotspot as the set containing all possible user actions that result in the same state transition. Examples of the definition of hotspots are presented in Fig. 2. Mouse events are included in hotspots as areas into which the pointer location fits. If widget information is available, as assumed in the RDP and ICA protocols, the widgets extents could



(a) Example graphical output.

(b) Corresponding FSM nodes and transitions.

Fig. 2: An example to illustrate the main concepts used throughout this paper. A user event causes frame buffer updates to be generated that form a transition to another state.

form the basis for creating hotspots. When only graphical primitives are supported, such as lines, rectangles and images, the coordinates of the containing primitive can be used as the defining region for the hotspot. When none of these higher level graphics can be intercepted, the system could record pointer event locations and use bounding boxes to construct hotspots. For key events, there is no coupling between the hotspot and graphical regions, as the hotspot then resolves to the key signature itself. A *corresponding hotspot* is defined as a hotspot that corresponds to user input, if in the case of mouse events the mouse location is inside the area defined by the hotspot. In the case of key strokes, a one-to-one mapping exists to the already registered keystrokes. When the viewer or server performs lookup of hotspots, this comprehension of correspondence is evaluated. A *matching hotspot* is defined as a corresponding hotspot with the additional requirement that the graphical updates closely resemble the actual application output.

4. Response time models

The sequence diagram presented in Fig. 3 indicates the latencies involved in our speculative remote display system. The responsiveness is measured as the time elapsed between the registration of a user action and the presentation of the results on screen. The software stack on the user's machine (consisting of operating system, virtualization layers, middleware, etc.) introduces an overhead delay between the actual user action and the registration of that action in the viewer application. For the sake of completeness, we have included these delays in our model, although these are hard to measure and typically negligible in comparison to network delays. Once the action is registered in the thin client viewer, the speculator predicts the display update and presents this on screen. Meanwhile, the thin client viewer encodes the user input for transfer over the network to the server. At the server, the user input is delivered both to the application that generates the actual screen output, and to the speculator that mimics the delivered prediction of the viewer, possibly searches for matching hotspots in other states and creates hotspots if necessary. The output of the speculator and the application are compared to decide how the client screen needs to be updated, as well as which changes are to be applied to the caching system (e.g., state merging, adding cache frames, adding hotspots) as detailed in Section 5. The graphics and other instructions for the viewer are encoded for transmission over the network. Then, at the viewer side, cache-specific processing will occur, to resynchronise the client with the server. After this step, the correctness of the speculated screen update is verified, to be corrected in case of a misprediction.

The metrics of interest are the latency measured between the user providing the input and both the first response as well as the final correct response presented on screen.

4.1. Traditional thin client protocol response model

Traditional thin client protocols have a response model as defined in Equation (1).

$$\delta_{traditionalTCproto} = \delta_{user-viewer} + \delta_{network} + \delta_{app} + \delta_{TCproto_{server}} + \delta_{TCproto_{viewer}} + \delta_{viewer-user} \quad (1)$$

with:

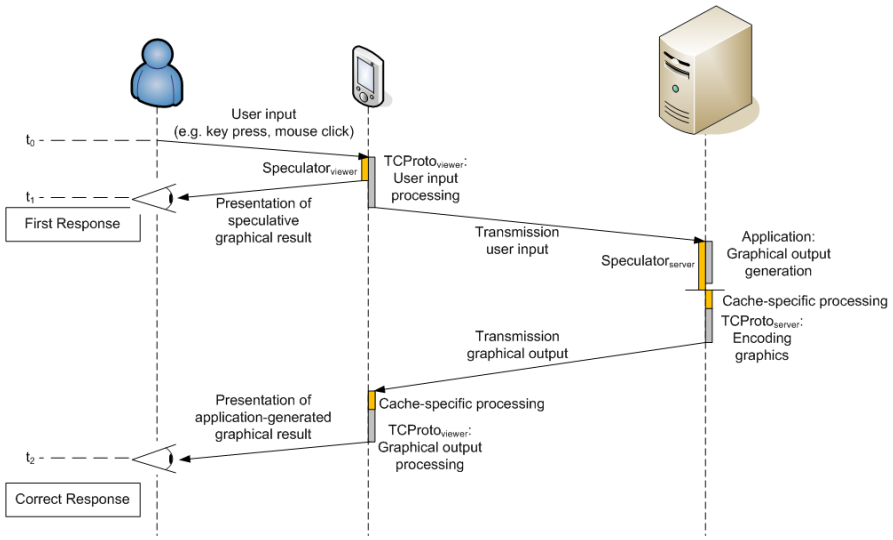


Fig. 3: Sequence diagram with latency terminology for the proposed speculative display system.

$\delta_{user-viewer}$ = the delay between the user providing input and the system registering the event

$\delta_{network}$ = the time spent for transmission of a user event and the responses over the network
$= \delta_{upstream} + \delta_{downstream}$

δ_{app} = the time needed for the application to generate the graphical responses

$\delta_{TCproto_{server}}$ = the delay at the server for the thin client protocol to deliver user events to the application and to encode graphical updates for transmission over the network

$\delta_{TCproto_{viewer}}$ = the delay at the viewer for the thin client protocol to forward user events over the network and to decode graphical updates for presentation on the user's screen

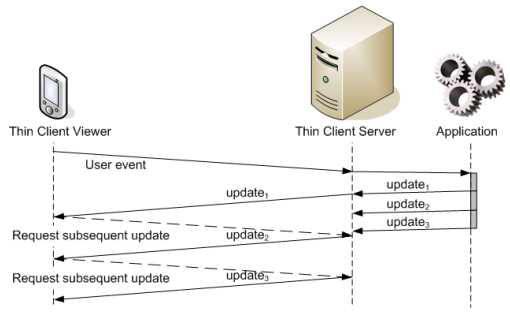
$\delta_{viewer-user}$ = the delay between the viewer software instructing the operating system to draw the frame buffer update and actual screen rendering

In this equation, $\delta_{network}$ can be broken down into subcomponents, depending on the particular thin client system. More specifically, thin client systems are based on *push* or *pull* protocols. The operation of a *pull* protocol such as the RFB protocol used in VNC is presented in Fig. 4(a). The first frame buffer update in a response to a user event is sent as soon as it is available at the server. The viewer requests subsequent frame buffer updates one at a time, on receipt of a frame buffer update. Therefor, $\delta_{network}$ in the case of a *pull* protocol requires one network RTT for each frame buffer update in the response to a user event, as presented in Equation (2a). The benefit of this request-based protocol is the inherent automatic network load balancing as explained in detail in [8]. In contrast, a *push* protocol, such as ICA and RDP, pushes the frame buffer updates as soon as they are available, without requests from the viewer. This flavour of thin client protocol is presented in Fig. 4(b). The advantage of this mechanism is that frame buffer updates in response to user events are delivered faster to the viewer. For these *push* protocols, one one-way upstream network delay is incurred to deliver the user event, and a one-way downstream network delay suffices for each frame buffer update to be delivered to the viewer, as modeled in Equation (2b).

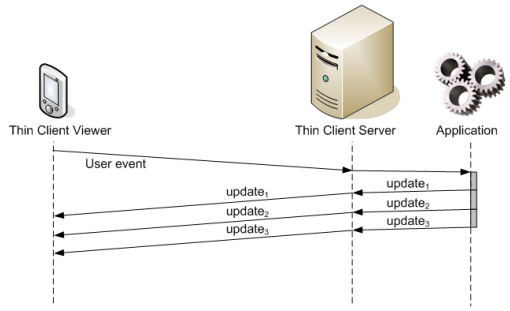
$$\delta_{network_{pull}} = N \times (\delta_{downstream} + \delta_{upstream}) + \delta_{userEvent} + \sum_{i=1}^{N} \delta_{update_i} \quad (2a)$$

$$\delta_{network_{push}} = N \times \delta_{downstream} + \delta_{upstream} + \delta_{userEvent} + \sum_{i=1}^{N} \delta_{update_i} \quad (2b)$$

with:



(a) Pull protocol.



(b) Push protocol.

Fig. 4: Different flavour of thin client protocols exhibit different operating schemes.

N = the number of frame buffer updates in the response to the user event
$\delta_{userEvent}$ = the transmission delay of the user event
δ_{update_i} = the transmission delay of frame buffer update i
$\delta_{downstream}$ = the downstream network delay
$\delta_{upstream}$ = the upstream network delay

4.2. First response

The delay until the first response is displayed on the screen ($\delta_{firstResponse}$), regardless of its correctness, indicates the optimal responsiveness that can be reached with the system. Although the correctness of this first response is not guaranteed, this metric plays an important role in the interactivity experienced by the user. The delay between the user input and the presentation of the first update depends on the speed of the viewer side speculator and the availability of a speculative response. If no matching hotspot is found in the current state, a network round trip is needed to acquire the first result coming from the server, as indicated in Equation (3).

$$\begin{aligned}\delta_{firstResponse} &= \delta_{user-viewer} \\ &+ P_{prediction} \times \delta_{speculator_{viewer}} \\ &+ (1 - P_{prediction}) \times [\delta_{network} \\ &+ \max(\delta_{app}, \delta_{speculator_{server}}) \\ &+ \delta'_{TCproto_{server}} + \delta'_{TCproto_{viewer}}] \\ &+ \delta_{viewer-user} \end{aligned} \quad (3)$$

with:

$P_{prediction}$ = the probability the user interaction can be mapped to hotspot in the cache
$\delta_{speculator_{viewer}}$ = the time required for the viewer speculator to acquire the speculative frame buffer updates
$\delta_{network_{userInput}}$ = the network delay for transmitting the user input from the viewer to the server
$\delta_{speculator_{server}}$ = the time required for the server speculator to acquire the speculative frame buffer updates
$\delta'_{TCproto_{server}}$ = the delay at server caused by the thin client protocol adapted to support speculative updating
$\delta_{network_{graphicalUpdate}}$ = the network delay for transmitting the frame buffer update from the server to the viewer
$\delta'_{TCproto_{viewer}}$ = the delay at the viewer caused by the thin client protocol adapted to support speculative updating

Here, the adaptations to the traditional thin client protocol denoted $\delta'_{TCproto_{server}}$ and $\delta'_{TCproto_{viewer}}$ are expressed with respect to the original thin client protocol (modeled in Equation (1)) as follows:

$$\delta'_{TCproto_{viewer}} = \delta_{TCproto_{viewer}} + \delta_{caching_{viewer}} \quad (4a)$$

$$\delta'_{TCproto_{server}} = \delta_{TCproto_{server}} + \delta_{caching_{server}} \quad (4b)$$

with:

$\delta_{caching_{viewer}}$ = the additional delay at the viewer, for synchronization of states and cache on request of the server
$\delta_{caching_{server}}$ = the additional delay at the server, caused by the algorithms involved to support speculative display updates, e.g., comparison of cached updates and actual application output or verification of correctness of the current state

4.3. Final correct response

The delay to display the final, correct graphical update, depends on the probability of correctly predicting the application output as described in Equation (5).

$$\begin{aligned}\delta_{correctResponse} &= \delta_{user-viewer} \\ &+ P_{correct|prediction} \times \delta_{speculator_{viewer}} \\ &+ (1 - P_{correct|prediction}) \times [\delta_{network} \\ &+ \max(\delta_{app}, \delta_{speculator_{server}}) \\ &+ \delta'_{TCproto_{server}} + \delta'_{TCproto_{viewer}}] \\ &+ \delta_{viewer-user} \end{aligned} \quad (5)$$

with:

$P_{correct|prediction}$ = the probability of correct prediction in case a prediction is effectively made

$1 - P_{correct|prediction}$ = the probability of making no prediction, or an incorrect prediction in case a prediction is made

Equation (5) is a generalization of the first response equation (Equation (3)). The first response delay model can be derived from this final correct response equation by omitting the correctness constraint ($P_{correct|prediction} = P_{prediction}$) and for $\delta_{network}$, to take only the first frame buffer update into account ($N = 1$).

Two extreme cases can be discerned, giving lower and upper bounds for the delay to present the final update, i.e.,

1. $P_{correct|prediction} \simeq 1$

$$\delta_{correctResponse} = \delta_{speculator_{viewer}} \quad (6)$$

2. $P_{correct|prediction} \simeq 0$

$$\begin{aligned}\delta_{correctResponse} &= \delta_{network} \\ &+ \max(\delta_{app}, \delta_{speculator_{server}}) \\ &+ \delta'_{TCproto_{server}} \\ &+ \delta'_{TCproto_{viewer}} \end{aligned} \quad (7)$$

4.4. Relation with the traditional thin client protocol

To study the performance of the speculative display system in comparison to the traditional thin client systems, the model for the traditional thin client protocol (Equation (1)) was substituted into the more generic final update equation (Equation (5)). The resulting model is presented in Equation (8).

$$\begin{aligned}&\delta_{speculativeTCproto} = \\ &P_{correct|prediction} \times (\delta_{user-viewer} + \delta_{speculator_{viewer}} \\ &+ \delta_{viewer-user}) \\ &+ (1 - P_{correct|prediction}) \times [\delta_{traditionalTCproto} \\ &+ \min(\delta_{speculator_{server}} - \delta_{app}, 0) \\ &+ \delta_{caching_{server}} + \delta_{caching_{viewer}}] \end{aligned} \quad (8)$$

This model identifies the components that incur additional latency for the speculative display system. Assuming that the viewer speculator responds faster than the network, the main components that could incur overhead latency are the server speculator and the caching provisions at the server and the viewer.

5. Speculative display algorithm

The logic for the speculative display system has been designed in an asymmetric way, such that the server controls the cache contents at the viewer side. Considering the use of battery-powered and resource-poor devices as a thin client viewer, the main advantage of this approach is that the viewer remains computationally simple.

5.1. Algorithmic functions common to viewer and server

The viewer and the server share functionality, to update the synchronized caches, to load frame buffer updates from the cache and to look up corresponding hotspots, as presented in Listing 1. Merging one state (denoted *oldState*) with another (denoted *newState*) involves three consecutive steps. First the target of incoming edges of *oldState* are changed to *newState* (lines 1 to 3). Next, the source of outgoing edges of *oldState* are altered to *newState* (lines 4 to 6). This way, *oldState* is separated from the FSM, and can be deleted (line 7).

Loading updates in a given state upon receipt of a user event occurs by searching for corresponding hotspots in this state (line 8). If a corresponding hotspot is found, the frame buffer updates registered with this hotspot are returned. Otherwise, none can be returned.

Lines 11 to 16 present how in a given state, corresponding hotspots are searched for given user input. As explained earlier, key signatures uniquely define a hotspot in a given state (lines 11 to 13). For mouse events, the location of the event must be enclosed in the hotspot's area to correspond (lines 14 to 16).

Listing 1 Functionality common to viewer and server

Function: mergeStates(*oldState*, *newState*)
1: **for all** *inEdges* ∈ *oldState*.incomingEdges **do**
2: *inEdge*.target = *newState*
3: **end for**
4: **for all** *outEdges* ∈ *oldState*.outgoingEdges **do**
5: *outEdge*.source = *newState*
6: **end for**
7: remove(*oldState*)

Function: loadUpdates(state *s*, event *e*)
8: **if** hasCorrespondingHotspot(*s*, *e*) **then**
9: **return** *correspHotspot*.updates
10: **end if**

Function: hasCorrespondingHotspot(state *s*, event *e*)
11: **if** *e*.type == *keyStroke* **then**
12: **return** *s*.hotspots.find(*e*.keySignature)
13: **end if**
14: **if** *e*.type == *mouseEvent* **then**
15: **return** *s*.hotspots.find(*e*.location ∈ hotspot.area)
16: **end if**

Listing 2 Viewer algorithmic functions

Function: onUserEvent(event *e*)
1: **if** hasCorrespondingHotspot(*currentState*, *e*) **then**
2: show(*correspHotspot*.updates)
3: *currentState* = *correspHotspot*.nextState
4: **else**
5: *currentState* = createNewState()
6: **end if**

Function: onGraphicalUpdate(Update *u*, cachedUpdateId *c*)
7: **if** $c \geq 0$ **then**
8: show(cache.getUpdate(*c*) + *u*)
9: **else**
10: show(*u*)
11: **end if**

5.2. Viewer algorithm

The algorithmic functions specific to the viewer, detailed in Listing 2, are straightforward. On receipt of a mouse event or a key stroke, the frame buffer updates that correlate to the corresponding hotspot in the current state are drawn on the screen. Accordingly, the viewer transitions to the next state (lines 1 to 3). If no corresponding hotspot is found in the current state, the response from the server is awaited and a new state is created (that has no hotspots) to avoid corrupt state transitions on subsequent user events (lines 4 to 6). Upon receiving a graphical response from the server, the viewer checks whether a valid cache update identifier is provided. If so, the graphical response represents a difference with respect to the cached update it needs to be added to (lines 7 and 8), otherwise, the update is intended to be presented on screen as is (lines 9 and 10). Other functionalities on the viewer, not included in Listing 2, are cache and FSM manipulations, that are merely applying these actions to elements indexed by the server.

5.3. Server algorithm

The algorithmic functions of the server are presented in Listing 3. As the application has no method to notify that a user event was completely processed (i.e., all graphical responses to the user event are generated), it is assumed that one user event marks the end of processing the previous user event (line 1 and lines 14 to 22 detailed later in this section). Lines 2 to 6 show that the frame buffer updates from corresponding hotspots, that can occur in any state in the FSM, are loaded for comparison with the application's graphical output once generated.

When graphical output of the application is available, the server evaluates the similarity with the frame buffer updates loaded earlier to decide whether the viewer has similar updates in cache. To this end, the function *isSimilar()* used in line 8 compares one set of frame buffer updates with another set of frame buffer updates. In our prototype, we implemented this function as to finding at least 2 'closely similar' updates in the sets, provided that the cardinality of the sets is at least 2. We defined the threshold for 'close similarity' to 90% of equal, overlapping pixels. If using this function, a closely similar update is found in the cache, the difference can be sent to save bandwidth, as described in lines 7 to 12. In the other case, the complete frame buffer update needs to be sent (line 13).

Lines 14 to 22 describe how a the completion of a state transition is handled. The transition is evaluated against the frame buffer updates from the corresponding hotspots loaded earlier (line 14). If close resemblance is found with the frame buffer updates of a hotspot other than the one the viewer had worked with, the server decides that the current state must be equivalent with the state the matching hotspot origins from (*matchingHotspot*.sourceState). As a result, both states are merged (lines 15 to 17). Merging of states is detailed in Section 5.1. The last step in handling a matching hotspot involves setting the current state to the target

Listing 3 Server algorithmic functions

```
Function: onUserEvent(event e)
 1: handleTransitionFinished(currentState, e)
 2: for all s ∈ states do
 3:     if hasCorrespondingHotspot(s, e) then
 4:         loadUpdates(s, e)
 5:     end if
 6: end for

Function: onApplicationOutput(Update u)
 7: for all hotspots ∈ correspHotspots do
 8:     if isSimilar(u, hotspot.updates) then
 9:         sendUpdate(update - u)
10:         return
11:     end if
12: end for
13: sendUpdate(u)

Function: handleTransitionFinished(State s, Event e)
14: if isSimilar(transition.updates, correspHotspot.updates) then
15:     if matchingHotspot != viewerHotspot then
16:         mergeStates(s, matchingHotspot.sourceState)
17:     end if
18:     currentState = matchingHotspot.targetState
19: else
20:     currentState = createNewState()
21:     createNewHotspot(s, e, currentState)
22: end if
```

state of the hotspot *matchingHotspot*.targetState (line 18). In case no matching hotspot is found, a new state is created to which the current state is set. Furthermore, a new hotspot is created from the user event, defining the transition between the previous state and the newly created current state (lines 19 to 22).

6. Architecture overview

The extension of traditional thin client systems to support the envisioned speculative display mechanism is shown in Fig. 5. Besides forwarding user events from the viewer to the server, they are delivered to the viewerside *cache handler*. This component completely handles the *onUserEvent()* viewer functionality presented in Section 5.2 that, using the *hotspot searcher* component, looks up whether a corresponding hotspot exists in the current state. If such a corresponding hotspot exists, the related frame buffer updates are fetched from the cache and displayed, even though at that moment,

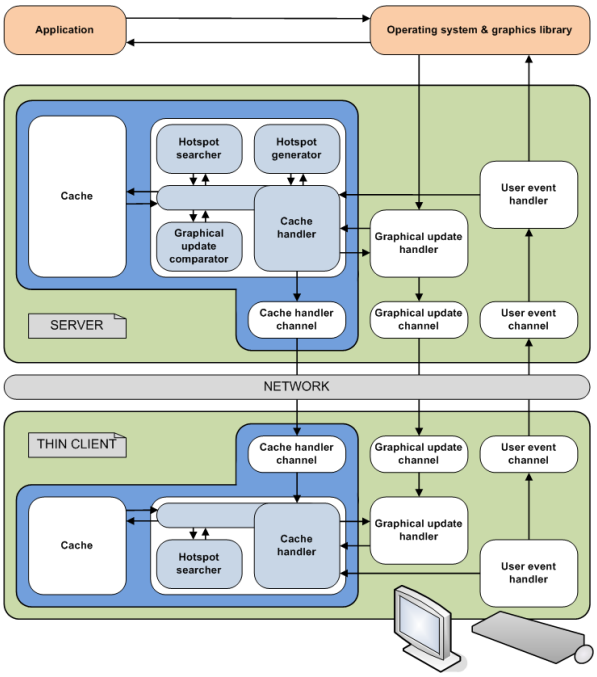


Fig. 5: Architecture for the proposed speculative remote display system. The additional components required for speculative display updating are marked by the blue highlight color.

the viewer has no absolute guarantee that this content is correct.

At the server, the received user input is delivered to the application. In parallel, the user input is delivered to the serverside counterpart of the *cache handler* that is triggered to handle the *handleTransitionFinished()* function described in Section 5.3. The server searches for *corresponding hotspots* in all states of the FSM using the *hotspot searcher* component. To evaluate whether the found hotspots are in fact *matching hotspots*, the *cache handler* relies on the *graphical update comparator* to evaluate the similarity between the application output and cached frame buffer updates. The *hotspot generator* facilitates the creation of new hotspots, based on the graphical content visible on screen at a given moment. The *cache handler* also handles *onApplicationOutput()* functionality, that requires the intervention of the *hotspot searcher* and *graphical update comparator* components to instruct the *graphical update handler* to either encode the current graphical application output directly or as a difference with respect to a cache frame buffer update.

The FSM is maintained in both *cache handler* components of the server and the viewer. Cache and FSM manipulation instructions from the server to the viewer are accomplished over the *cache handler channel*.

7. Experimental results

7.1. Setup

For the experiments, one single machine hosted both server and client in order to keep full control over the network. An AMD Athlon™ 64 X2 Dual Core Processor 6000+, 1 GHz machine with 2 GB RAM was used. The localhost network was used, with the Linux traffic control *tc* impairment tool. Using this tool, network latencies were configured. However, the bandwidth remained uncapped resulting in negligible transmission delays, i.e., no additional latency is incurred depending on the size of frame buffer updates or user events. As a result, in the definition of $\delta_{network}$ in Equation (2a) and Equation (2b), $\delta_{userEvent}$ and δ_{update_i} can be neglected. The Linux *XMacro* package was used to record and replay the user events of the selected actions for the experiment scenarios. We used TigerVNC (version 1.2.0) [16] as the basis for implementation of our speculative system. The screen resolution was configured to 1280 $\times$ 720.

7.2. Test scenario

The speculative display mechanism has been tested using the text editor *gedit* [17], for which sixteen actions were recorded. These actions consisted of opening the different menus in the application by clicking on the menu and closing it by clicking the menu again. This yields seven actions, for the menus 'File', 'Edit', 'View', 'Search', 'Tools', 'Documents' and 'Help'. For each of these actions, an alternative action was defined to open the menu the same way, but closing it by clicking in an unrelated area in the application. The final two actions were obtained by opening the 'Help' menu, selecting the 'About' item, and closing the corresponding dialog box either using the 'Close' button or closing with the 'X' button in the window decorator. This set of actions was specifically defined as series of mouse events that cause a transition from a given state to itself over other states. For example, the filemenu open-close action exists of a mouseclick on the 'File' menu, leading to the expansion of the menu as a second state, followed by a second mouseclick on the file menu causing the collapse of the menu returning to the initial state. This approach allows creating random scenarios by randomly selecting actions from this set, with equal probability. For our experiments, we have created scenarios by uniformly distributed random drawing of actions from this set.

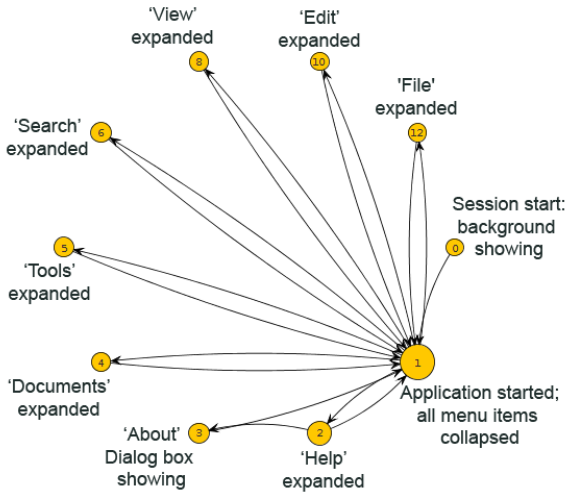


Fig. 6: Generated Finite-State Machine for executing 100 actions selected from 16 actions randomly according to a uniform distribution.

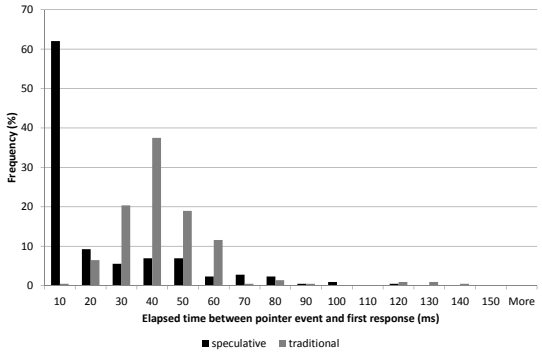
7.3. Results

7.3.1. Generated Finite-State Machine

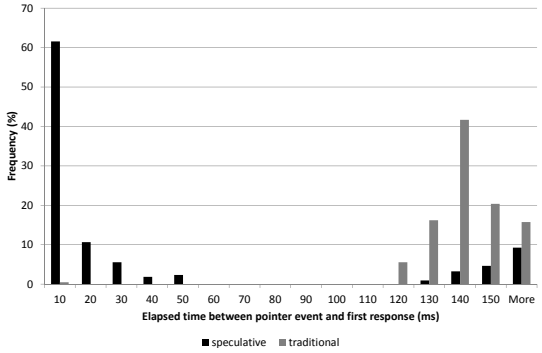
The FSM presented in Fig. 6 was obtained by executing 100 actions randomly drawn from the set of 16 actions. In total, 216 user events were registered. The system has recognized the 7 different application menus, for which the expansion and collapse leads to a transition to state 1. The expansion of the menu leads to a new state, while the collapse results in a return to the original state. The alternative collapse methods are not visible in this FSM, as these actions are contained in the same hotspot and hence result in one transition. The figure also shows that the presentation of the 'About' dialog box (in state 3) from the 'Help' menu (state 2) is recognized as having expanded the 'Help' menu first as a transition over state 2 is performed. The transition from state 0 to state 1 represents the start of the *gedit* application.

7.3.2. First response

Figure 7 compares the responsiveness of the speculative system to the traditional thin client system, measured as the time elapsed between the user event and the first response shown on screen, irrespective of the correctness in comparison to the real application output. Figure 7(a) shows the impact when no network latency is configured. Both systems respond within 150 ms at all times, but while the traditional system responds in less than 40 ms for 64.81% of the user events, the speculative system accomplishes this for 83.79% of the user events. On average, the traditional system responds in



(a) No network latency.



(b) 50 ms network latency.

Fig. 7: Comparison of first responses for the traditional thin client protocol and for the speculative display mechanism.

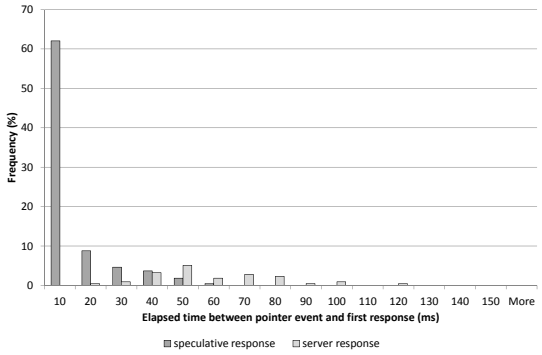
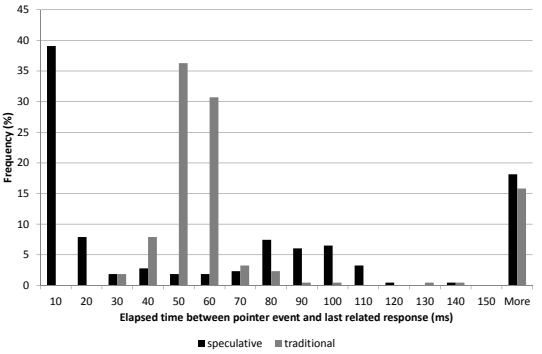
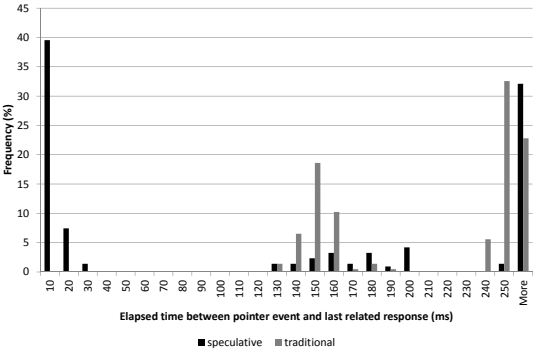


Fig. 8: No network latency; breakdown of the first responses of the speculative display mechanism into speculative responses and server responses.



(a) No network latency.



(b) 50 ms network latency.

Fig. 9: Comparison of correct responses for the traditional thin client protocol and for the speculative display mechanism.

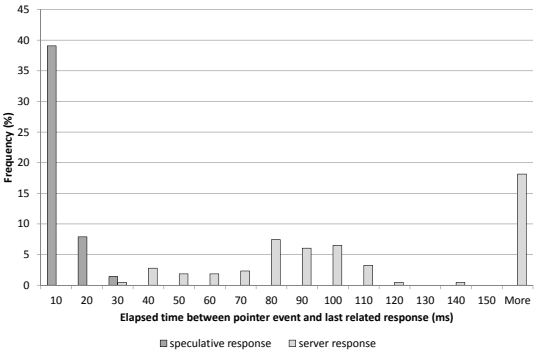


Fig. 10: No network latency; breakdown of the correct responses of the speculative display mechanism into speculative responses and server responses.

38.79 ms, while in the speculative system the average decreases to 18.15 ms. The standard deviation however increases from 17.48 ms to 21.78 ms, indicating that more jitter is introduced.

The contrast between the speculative display system and the traditional thin client system is more apparent in Fig. 7(b), for a network latency of 50 ms, resulting in a minimum network RTT of 100 ms. It shows that speculative responses are independent of the network latency, as in this experiment 79.62% of the user events were responded to in less than 40 ms. With the traditional system, first responses to user events were delivered in less than 150 ms in 84.26% of the cases, with the speculative system this happens in 90.74% of the cases.

Figure 8 presents the breakdown of the experiment results with no network latency configured. It shows

the share of speculative responses and server responses in the first responses measured for the speculative display system in Fig. 7. Speculative responses are delivered in 9.73 ms on average, while responses from the server arrive after 55.19 ms. In the experiment with 50 ms one-way network latency configured, the server responses arrive after 154.49 ms on average. Over the 216 user events in the session, we have recorded 176 speculative display updates, resulting in a prediction probability $P_{prediction}$ of 81.48%. However, this includes the learning stage of the system. When assuming regime operation in the last half of the experiment session, 115 of the 117 user events (98.29%) resulted in a speculative update of the viewer's display.

7.3.3. Correct response

The first response gives an indication of the reactivity of the system. However, it is equally important to evaluate the performance of the speculative system in terms of the speedup acquired in presenting the correct response on screen.

Figure 9 shows the times elapsed between the user event and the pristine viewer screen content. Figure 9(a) shows the impact when no network latency is configured. Where the traditional system can bring the viewer to a pristine state in less than 40 ms for only 9.76% of the user events, the speculative display system achieves this for 51.63% of the user events. The fact that for the traditional VNC viewer used in the experiments, the latency exceeds 40 ms in more than 90% of the cases can be explained by the implemented optimization of deferred updates at the server, that uses a fixed waiting time of 40 ms in which graphical updates are joined for collective transmission to the viewer. Eventhough, this optimization is also active in the speculative version that uses the same VNC implementation as a basis for the prototype. Figure 9(b) presents results from the session with 50 ms network latency configured. In this experiment, the traditional system displays the pristine state in less than 150 ms for 26.51% of the user events. For the speculative system this is obtained for 53.49%. The traditional system is unable to present the correct screen state in less than 120 ms, while the speculative system achieves this in 48.37% of the user events.

In our prototype, VNC is used as the base system for implementation which relies on a *pull* protocol. The latency for presenting the complete graphical update is heavily influenced by the number of frame buffer updates it consists of. As mentioned in Equation (2a) in Section 4.1, a network RTT is required for each frame buffer update, which explains the occurence of a large amount of responses arriving after more than 150 ms and 250 ms for the respective different network latencies configured. Also, application developers and graphical libraries often deliberately incorporate delays between updating GUI elements to increase the usability of the application, indicating that some of the responses in the 'More' category are less harmful for the user experience than the figure suggests.

Figure 10 presents the breakdown of the experiment results with no network latency configured. It shows the share of speculative responses and server responses in bringing the viewer to a pristine state, measured for the speculative display system in Fig. 9. When the screen state is correctly updated by the viewer speculator, this is accomplished in 7.35 ms on average. If the server needs to interfere, the network latency causes on average 265.54 ms latency between the user event and presenting the complete graphical output on screen. Over the complete session, 104 of the user events were correctly handled by the speculator, resulting in a probability for correct prediction of $P_{correct|prediction}$ 48.37%. Again, considering the first half of the session as learning phase, a probability of 70.69% is obtained.

7.3.4. Bandwidth reduction

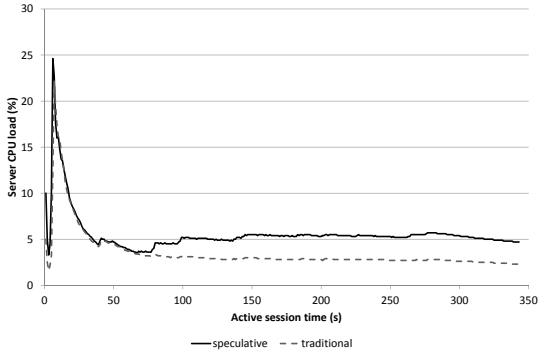
We configured the system to transfer graphical updates in raw, uncompressed pixel format to avoid distortion of the bandwidth measurements due to the use of specific compression algorithms. The bandwidth consumed in the session for both the speculative and the traditional system is presented in Table 1. As the startup of the session, i.e., loading the desktop background, and the startup of the application could not be predicted, we have also shown the values after this initialization. The table shows that the cache used in the speculative system allows to reduce the consumed bandwidth in comparison to the traditional system. The amount of this reduction depends on the prediction accuracy, the frequency of predictable user events in the session, the graphical update size of the transitions between states to name a few. In our specific experiment, roughly 50 MB was saved by using the speculative system.

7.3.5. Algorithm overhead

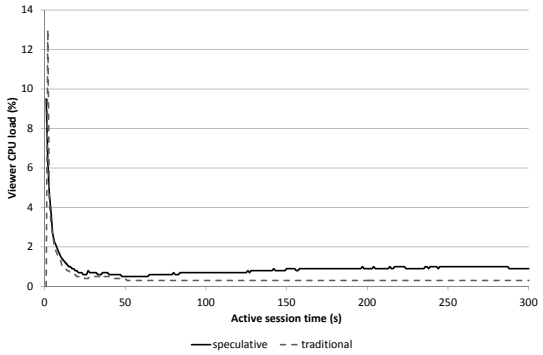
The computational overhead was measured by periodically logging the CPU usage during the execution of the test scenarios. The values were acquired using the Linux *ps* command line tool. Figure 11 shows that the computational overhead is limited. In its regime condition, i.e., after a peak load when the application is started in the session and the viewer connection is completed, the computational overhead of the speculative display algorithm amounts to 2% for the server, as

Table 1: Bandwidth consumed in the experiment session (uncompressed raw pixel format, averaged over 10 iterations)

	network latency	complete session	without initialization
speculative	0 ms	26252 kB	19443 kB
	50 ms	27661 kB	20858 kB
traditional	0 ms	76482 kB	69654 kB
	50 ms	75852 kB	69047 kB



(a) Server.



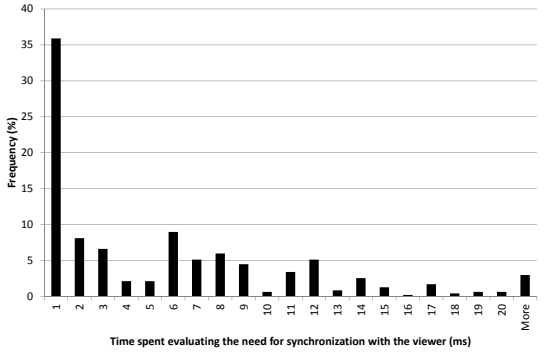
(b) Viewer.

Fig. 11: CPU load comparison of the speculative mechanism and the traditional thin client system.

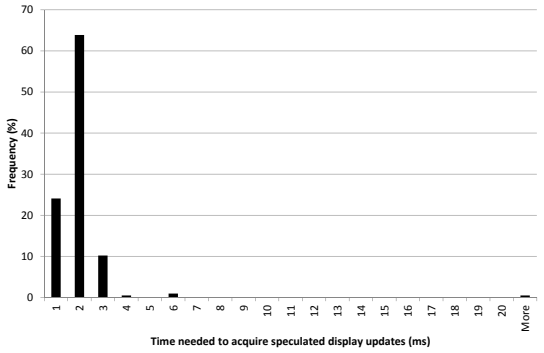
shown in Figure 11(a), and less than 1% for the viewer as shown in Figure 11(b).

Concerning synchronization messaging the overhead is also very limited. For the complete session with 217 user events reacted upon, only 2653 bytes were sent from server to viewer to maintain synchronized FSM and caches. Compared to the graphical content that needs to be transfered over the network (about 27 MB in raw format as indicated in Table 1, this synchronization overhead is considered negligible.

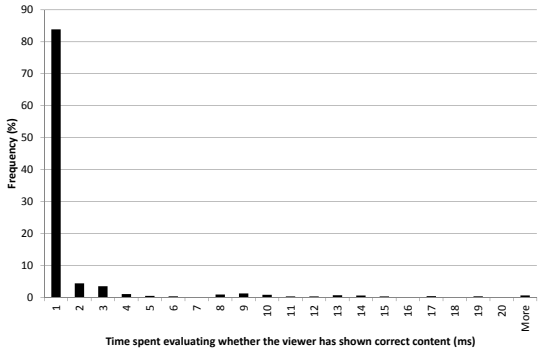
The speculative system incurs overhead in comparison to the traditional, unaltered thin client system. The most frequently used functionalities were monitored, from which the results are presented in Fig. 12. The



(a) Time spent evaluating the need for synchronization with the viewer.



(b) Time needed to acquire the speculated display updates.



(c) Time spent evaluating whether the viewer has shown correct content.

Fig. 12: Additional latency incurred by selected, frequently used functionality of the speculative display algorithms.

histogram of the time spent evaluating the need for synchronization with the viewer, shown in Fig. 12(a), exhibits a rather fluctuating behaviour. The average of these overhead values amounts to 5.38 ms. Figure 12(b) presents the time needed to acquire speculated display updates. As this concerns a simple search for a hotspot in one state, and loading the frame buffer updates, this task finishes within an order of a couple of milliseconds, with an average of 1.24 ms. When transfering graphical updates from the server to the viewer, the server evaluates whether the viewer had correctly updated the screen speculatively, and if not, whether similar content is available in the viewer's cache to apply difference encoding. Figure 12(c) shows that the additional latency incurred by this functionality is limited, with an average of 1.46 ms.

8. Conclusions and future work

In this paper a mechanism is proposed to augment remote application rendering in the cloud by speculatively displaying application output to the user, in spite of creating an impression for the user that the underlying network latency does not influence the reactivity of the system. The experimental results show that the reactivity of the system is improved. In our experiments, a first response to a user event, irrespective of the correctness, is shown on screen within 40 ms for over 80% of the user events. After the learning phase in which the FSM was derived and hence less accurate predictions were made, 70.96% of the user events were correctly responded to by the speculator. The results also show that little overhead is induced by the system, both concerning overhead synchronization messaging and server and viewer CPU load amounting to 2% and less than 1% CPU load respectively. The network load caused by the synchronization of the caches of the server and the viewer was shown to be negligible in comparison to the graphics that need to be transmitted, as in our experiment, just over 2.6 kB were spent on such messages. When the viewer is unable to predict the application output correctly, the server needs to interfere. The most complex task for this mechanism was shown to be the evaluation whether server-viewer synchronization is needed, requiring to compare the application output to the frame buffer updates in the cache. The average overhead latency caused by this component amounts to 5.83 ms.

If in contrast to the assumptions in this paper, limited storage on the viewer is considered, the choice of items to store and the ratio between hotspots, states and frame buffer updates needs to be optimized. To this end, graph cutting algorithms in combination with usage tracking of states and hotspots is a promising route for future investigation. Storing the FSM, cache items and hotspots belonging to applications over different user sessions might be an option to take into consideration. Since applications are expected to look and behave the same for different users and over different execution times of the application, it could be interesting to store these globally, and to be loaded at the start of a session. However, private data should be removed, which could be accomplished by retaining only exact matches between users. With increasing network latencies, one would intuitively expect larger benefits from the speculative display system as a correct prediction results in large latency reduction. However, in this case, errors are shown on screen for a considerable time during which the user can interact. On correction of the erroneous content, the user might get confused. A viable solution is to limit the number of unvalidated updates to mitigate the desynchronization between speculative and actual application output, but the introduced jitter in speculative display updates might result in an unacceptable user experience. As part of future research, we see opportunities to evaluate the impact of prediction accuracy and network latency on Quality of Experience (QoE) through subjective assessment.

Acknowledgement

Bert Vankeirsbilck is funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Part of this work has been funded by the UGent GOA project "Autonomic Networked Multimedia Systems".

References

[1] A. Lai, J. Nieh, Limits of wide-area thin-client computing, in: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, ACM, New York, NY, USA, 2002, pp. 228–239. doi:10.1145/511334.511363.

[2] A. Lai, J. Nieh, On the performance of wide-area thin-client computing, ACM Trans. Comput. Syst. 24 (2) (2006) 175–209. doi:10.1145/1132026.1132029.

[3] N. Tolia, D. Andersen, M. Satyanarayanan, Quantifying interactive user experience on thin clients, Computer 39 (3) (2006) 46 – 52. doi:10.1109/MC.2006.101.

[4] M. Jovic, M. Hauswirth, Measuring the performance of interactive applications with listener latency profiling, in: Proceedings of the 6th international symposium on Principles and practice of programming in Java, PPPJ '08, ACM, New York, NY, USA, 2008, pp. 137–146. doi:10.1145/1411732.1411751.

[5] R. Sharp, Latency in cloud-based interactive streaming content, Bell Labs Technical Journal 17 (2) (2012) 67–80.

doi:10.1002/bltj.21545.
URL http://dx.doi.org/10.1002/bltj.21545

[6] B. Vankeirsbilck, P. Simoens, J. De Wachter, L. Deboosere, F. De Turck, B. Dhoedt, P. Demeester, Bandwidth optimization for mobile thin client computing through graphical update caching, in: Australasian Telecommunication Networks and Applications Conference (ATNAC), 2008, pp. 385 – 390. doi:10.1109/ATNAC.2008.4783355.

[7] J. R. Lange, P. A. Dinda, S. Rossoff, Experiences with client-based speculative remote display, in: USENIX 2008 Annual Technical Conference on Annual Technical Conference, ATC'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 419–432. doi:1404014.1404048.

[8] T. Richardson, Q. Stafford-Fraser, K. R. Wood, A. Hopper, Virtual network computing, IEEE Internet Computing 02 (1) (1998) 33–38. doi:10.1109/4236.656066.

[9] Microsoft Corporation, Windows Remote Desktop Protocol (RDP), http://msdn2.microsoft.com/en-us/library/aa383015.aspx.

[10] A. Wasserman, Extending state transition diagrams for the specification of human-computer interaction, Software Engineering, IEEE Transactions on SE-11 (8) (1985) 699–713. doi:10.1109/TSE.1985.232519.

[11] V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, P. Jones, Model-based testing of gui-driven applications, in: S. Lee, P. Narasimhan (Eds.), Software Technologies for Embedded and Ubiquitous Systems, Vol. 5860 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 203–214. doi:10.1007/978-3-642-10265-3_19.

[12] A. M. Memon, B. N. Nguyen, Advances in automated model-based system testing of software applications with a gui front-end, in: M. V. Zelkowitz (Ed.), Advances in Computers, Vol. 80 of Advances in Computers, Elsevier, 2010, pp. 121 – 162. doi:10.1016/S0065-2458(10)80003-8.

[13] A. Memon, I. Banerjee, A. Nagarajan, GUI ripping: Reverse engineering of graphical user interfaces for testing, in: proceedings of the 10th working conference on reverse engineering (WCRE03), Vol. 1095, Citeseer, 2003, pp. 260–269.

[14] L. Zhong, N. K. Jha, Dynamic power optimization targeting user delays in interactive systems, IEEE Transactions on Mobile Computing 5 (11) (2006) 1473–1488.

[15] M. Sumalatha, S. Sridhar, G. Satish, A novel thin client architecture with hybrid push-pull model, adaptive display prefetching and graph colouring, International Journal of Ad hoc, Sensor and Ubiquitous Computing (IJASUC) 3 (3) (2012) 67 – 77. doi:10.5121/ijasuc.2012.3305.

[16] TigerVNC, www.tigervnc.org.

[17] The GNOME Project, gedit, http://projects.gnome.org/gedit/.