2009 29th IEEE International Conference on Distributed Computing Systems

An Approach to Sharing Legacy TV/Arcade Games for Real-Time Collaboration

Sili Zhao†, Du Li‡, Hansu Gu†, Bin Shao†, Ning Gu†
†School of Computer Science, Fudan University, China
‡ Nokia Research Center, Palo Alto, California, USA
Contact Email: †ninggu@fudan.edu.cn, ‡lidu008@gmail.com

Abstract

Interactive TV/arcade games have been entertaining people for over 50 years. Nowadays a large number of legacy TV/arcade games have been ported to new platforms such as PCs by emulation. However, they generally require that the players be co-located to interact with one computer that emulates the game. This paper proposes a novel approach to turning those single-computer games into multi-computer games such that multiple players can play their favorite legacy games in real time over a computer network. The main challenge in this work is how to synchronize multiple replicas of a game without semantic knowledge about or modifications to the game. We present the approach, especially the synchronization algorithm, and evaluate its effectiveness under a variety of network conditions. In future research we will extend this work on mobile devices.

1. Introduction

Interactive TV and arcade games have a long history which can be traced back to Tennis for Two in 1958. Since then gaming has grown into a significant part of the entertainment industry. Over the recent two decades alone, hundreds of millions of TV/arcade game consoles have been deployed. Thanks to modern software techniques such as virtual machines (VMs), now we can also play TV and arcade games on personal computers. A large number of game platforms can be emulated on PCs, such as FC,[1] SFC,[2] Nintendo 64, MD,[3] and PS.[4] In principle, all games developed for a platform can be played on PCs if there is a VM for that platform. The number of games that can be emulated has already reached over tens of thousands according to incomplete statistics from the Internet. [5]

However, in general, existing game VMs to the best of our knowledge require that players of an emulated game be co-located. The players usually use different input devices to interact with the same computer that emulates the game. This requirement is not necessarily a limitation in situations where the players can get together conveniently. Nevertheless, it could have been relaxed with modern collaboration technologies so that people can play their favorite games in real time without having to be physically co-located.

In this paper, we present a novel technique that can turn single-machine TV/arcade games into distributed multi-user games that can be played by multiple users over a computer network. The approach, called *real-time collaboration transparency*, extends the (single-computer) game virtual machines with a novel consistency control algorithm such that replicas of the extended VMs running on different computers can be synchronized in real time. The approach is "game transparent" as it does not need to modify the games themselves nor acquire semantic knowledge about their workings. As a result, the myriad of legacy TV/arcade games can be automatically turned into distributed games once their VMs are extended with our approach.

The technical challenge in our approach is mainly that the distributed game VM not only has to guarantee convergence of data replicas but also has to achieve it in real time and, moreover, must be transparent to the original games. By comparison, TV/arcade games and their single-computer emulations do not need to consider synchronization; traditional collaboration transparency approaches, e.g., [1, 6, 7, 8, 14], which aim to convert single-user software applications into multi-user applications without modifying source code, in general do not have the real-time constraint; specialized multi-player games, e.g., [4, 5, 12, 9, 3], on the other hand, do not have the "game transparency" constraint and hence their synchronization techniques in general cannot be applied in this context.

The rest of this paper is as follows. Section 2 overviews the system design. Section 3 presents the synchronization algorithm. Section 4 discusses performance experiments. Section 5 compares related research. Section 6 concludes.

[1] Famicom http://en.wikipedia.org/wiki/Nintendo_Entertainment_System
[2] Super Famicom http://en.wikipedia.org/wiki/Super_Family_Computer
[3] Mega Drive http://en.wikipedia.org/wiki/Megadrive
[4] Play Station http://en.wikipedia.org/wiki/Play_Station
[5] `http://www.allgoodthings.us`

1063-6927/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDCS.2009.26

2. Overview of System

Most TV/arcade games to our knowledge only allow for two players. Therefore our system and the synchronization algorithm assume two sites in this paper. Suppose that two users want to play a TV/arcade game on two networked computers. They must install our extended VM and the same game image. Some rendezvous mechanism is required for them to find each other, such as instant messenger and games lobby. Then a UDP-based communication channel will be established. After that, the game image is loaded onto the two VMs and the users can use the input devices on their own computers to play the same game together.

Our extended VM inherits the original virtual modules that emulate hardware behavior of the source TV/arcade platform, such as virtual audio/video, input devices, CPU, and memory. They are driven by user inputs to perform state transitions and generate source platform dependent audio/video outputs. The VM translates the game outputs into target platform dependent outputs to the physical audio/video devices and present them to the players. The input module reads user inputs from input devices on the target platform and feeds them to the virtual input module.

These modules are reused in our system. We introduce a new sync module that implements logical consistency and real-time consistency, as will be presented in detail in the next section. In the original VM, the user inputs are committed directly to the virtual input module, whereas in our work inputs from different sites must be synchronized first before they are committed so as to achieve consistency.

Most open-source game VMs to our knowledge can be extended with minor modifications. For example, our system is adapted from the MAME virtual machine source code. [6] The two new modules take only about 800 lines of C code in total. Therefore, due to its simplicity, we will not go into further details of the system implementation. Instead, we will focus on the critical logical and real-time consistency control algorithms underlying the sync module.

Algorithm 1 gives the pseudo code of how our distributed game virtual machine works. It is the same as its single-machine version except the three underlined steps 5, 7, and 10. These three steps implement synchronization in our work. When there are two sites, the same initial state S is replicated. Inputs are obtained at the two sites (step 6), respectively, and merged (step 7) before fed into the game (step 8). The main idea for achieving logical consistency is local lag [9], i.e., the local inputs are buffered until remote inputs to the same frame are received and then they are passed to the game together (step 7). The main idea for achieving real-time consistency is to synchronize the pace of every frame (steps 5 and 10 combined).

[6] `http://mamedev.org/`

Algorithm 1 Pseudocode of distributed VM

```
1:  var State S = initial state;
2:  var Input I = 0;
3:  var Integer Frame = 0;
4:  repeat
5:    BeginFrameTiming();
6:    I = GetInput();
7:    I' = SyncInput(I, Frame);
8:    S' = Transition(I', S);
9:    translate and present S';
10:   EndFrameTiming();
11:   Frame++;
12: until end of game;
```

In the step of $S' = \text{Transition}(I, S)$, the game itself makes a state transition based on its current state S and the supplied input I. Notation S' is just another view of S which could be assigned to S. It is worth noting that state transition is a black box to this work. We do not seek to modify the game behavior nor sneak into the game itself to figure out how it works. The game will not be aware of fact it is played on multiple computers connected by a digital network. That is, our work is "transparent" to the game, and the game is also transparent to the extended VM.

3. The Synchronization Algorithm

For multiple players to play the same game, it is critical to maintain consistency between replicated states at all involved sites. Consistency maintenance in this context includes two aspects: logical consistency and real-time consistency. Traditional distributed systems in general only ensure *logical consistency*, or convergence of states. In addition to the requirement for convergence, multimedia collaboration systems, such as multiplayer online games, often need to ensure *real-time consistency*, i.e., different sites must produce the same sequence of output states and reach each state within some human-tolerable bound.

For convenience of discussions, we extend the variables in Algorithm 1 as follows: Frame[k] denotes the current frame at site k; I[k, f] denotes site k's input to frame f; and S[k, f] denotes site k's state corresponding to frame f. In a distributed VM, one site only contributes a portion of input variable I; we view the input as a binary string, in which different sites control different bits of the string. Notation SET[k] maps site k to the set of bits it controls. For any two different sites j and k, SET[j] $\cap$ SET[k] = {}. Notation $I[j, f](\text{SET}[k])$ denotes site k's bits in the input to frame f at site j. The bits that are not controlled by any sites are mapped to SET[-1] and ignored in processing.

We assume that the original game VM is deterministic, i.e., with the same initial state and same input sequence,

the VM always produce the same sequence of output states. Under this assumption, consider a system with two sites that start from the same initial state. For every frame, if the two sites have the same partial input from each other site, the VM always produce the same sequence of output states.

This is the condition for logical consistency. If this condition is satisfied, the VM will produce the same sequence of output states and the same final state at all sites in the system. In order to achieve so, the sites need to exchange partial inputs. In a simplest scheme, each site broadcasts its partial inputs to all other sites, and integrates all received partial inputs to every frame. Then the integrated inputs are fed to the frame. To ensure the same inputs to every frame, each site must delay the execution of its local inputs until all partial inputs to that frame are received.

The idea of delaying the execution of local inputs is called "local lag", which trades off local responsiveness temporarily to mitigate inconsistencies across sites [9]. Intuitively, local lag must be a small value because a human player usually expects to see the effects of her own inputs in a short time. It has long been established in the HCI field that local response time should not go beyond 100 ms [13]. Interestingly, studies of distributed multimedia systems also reveal that, to achieve desirable interactivity, end-to-end delay should not go beyond 100 ms [4, 10, 11].

Therefore, we recommend that our system, or any system that takes a similar approach to sharing legacy TV/arcade games, should be used in networking environments with one-way latencies no longer than 100 ms. Here we emphasize one-way latencies because it takes at least a single trip for a partial input to arrive at a remote site. Note, however, that 100ms is a subjective order of magnitude rather than definitive value, which may vary with players and other conditions [4, 10, 11, 13].

3.1. Algorithm for Logical Consistency

To achieve logical consistency, we replicate the game image to ensure that the VMs start from the same inital state; our logical consistency control algorithm ensures the same input to every frame. Over a packet-switching network, there are problems such as packet loss, packet disorder, and packet duplication. As a reliable transport, TCP solves those problems. However, it is problematic in satisfying the real time constraint. Therefore, like in many other realtime applications, we resort to UDP and implement some of the reliability mechanisms in TCP.

As in Algorithm 1, we ensure the same input condition in function I' = SyncInput(I, F), where I is the local input on frame F and I' is the synchronized input on frame F that combines the local and remote inputs on the same frame. Algorithm 2 shows the pseudocode of SyncInput. In this paper, we assume there are only two sites in our system. The algorithm defines the following five constants.

- **MySiteNo**: the local site number, 0 or 1.
- **RmSiteNo**: the remote site number, 0 or 1.
- **MySET**: the local input bits SET[MySiteNo].
- **RmSET**: the remote input bits SET[RmSiteNo].
- **BufFrame**: the local lag value (number of frames).

The algorithm uses the following four global variables.

- **IBuf**: the input buffer (array), all elements initialized to zero. IBuf[f](SET[i]) denotes the partial input of site i for frame f. A buffer of unlimited size is assumed here for simplicity in presentation.
- **IBufPointer**: initialized to zero, the pointer points to the element in IBuf that should be returned from current call to the SyncInput algorithm.
- **LastAckFrame**: a two-element array with both elements initialized to (BufFrame-1). LastAckFrame[MySiteNo] is not used in a two-site algorithm. LastAckFrame[RmSiteNo] is the frame number of the last local partial input that has been acknowledged by the remote site. This suggests that the remote site has successfully received all my partial inputs up to frame LastAckFrame[RmSiteNo].
- **LastRcvFrame**: a two-element array with both elements initialized to (BufFrame-1). LastRcvFrame[i] is the last frame up to which the partial inputs from site i have been received and filled into IBuf. An element of IBuf is delivered by SyncInput only when partial inputs from both sites have been received for that frame.

In Algorithm 2, lines 1-5 buffer the local partial inputs I. When I for frame F is received, it is delayed by $BufFrame$ frames. The number is calculated to match the local lag time of around 100 ms. If the expected frame rate is 60 FPS, [7] or 16.7 ms per frame, $BufFrame$ can be set as 6. Hence the local input is added to the input buffer for frame $LagF = F + 6$ instead of that for frame F. Since the SyncInput algorithm is called for every frame, F = 0, 1, 2, ..., the local partial inputs are added into $IBuf$ continuously but with the first 6 frames skipped. That is, the players will not see effects of their inputs until after the first 6 frames or 100 ms.

The loop of lines 6-21 can be divided into two parts. The first is to send a sync message sd to the remote site (lines 7-11) and the second is to receive a sync message rc from the remote site (lines 12-20). Message sd is to notify the remote site of local partial inputs (sd[3...]) that have not been acknowledged and to acknowledge receipt of remote partial inputs up to some frame (sd[0]). Accordingly, when a remote message rc is received, the received remote partial inputs are extracted to update local input buffer (line 13); the LastRcvFrame variable is updated (lines 14-16), and so is the LastAckFrame variable (lines 17-19).

[7] This is because most arcade games to our knowledge are 60 FPS.

Algorithm 2 Pseudocode of SyncInput(I, F)

```
1:  var Integer LagF = F + BufFrame;
2:  if LastRcvFrame[MySiteNo] < LagF then
3:     IBuf[LagF](MySET) = I(MySET);
4:     LastRcvFrame[MySiteNo] = LagF;
5:  end if
6:  repeat
7:     send message sd to remote site if new info exists:
8:           sd[0] = LastRcvFrame[RmSiteNo],
9:           sd[1] = LastAckFrame[RmSiteNo]+1,
10:          sd[2] = LastRcvFrame[MySiteNo],
11:          sd[3...] = IBuf[sd[1]](MySET)
                        ...IBuf[sd[2]](MySET);
12:    if received message rc from remote site then
13:       Update IBuf[rc[1]](RmSET)...IBuf[rc[2]](RmSET)
                by received remote partial inputs rc[3...];
14:       if rc[2] > LastRcvFrame[RmSiteNo] then
15:          LastRcvFrame[RmSiteNo] = rc[2];
16:       end if
17:       if rc[0] > LastAckFrame[RmSiteNo] then
18:          LastAckFrame[RmSiteNo] = rc[0];
19:       end if
20:    end if
21: until LastRcvFrame[RmSiteNo] ≥ IBufPointer
22: IBufPointer++ ;
23: return IBuf [IBufPointer-1];
```

The loop exit condition in line 21 is to ensure that the current input variable will not be returned until the remote input for the same frame has been received. Hence all inputs to one frame are executed before proceeding to the next frame. This way the same input sequence is achieved at both sites. Implicitly, the causal ordering of input events are also maintained. As a result, the same sequence of output states as well as the same final state are produced at both sites.

Note that, for the first six frames, the exit condition is trivially satisfied and empty inputs are returned. When F=6, if the remote partial inputs to the actual first frame is received (or those to the first few frames are received), LastRcvFrame[RmSiteNo] ≥ 6 must hold, then the exit condition is satisfied and the (complete) input to the actual first frame is returned; otherwise, LastRcvFrame[RmSiteNo] would remain in its initial value 5 and the loop would not exit.

When partial input to some frame of one site is lost in transmission, acknowledgement to that frame will not be received by the other site. Hence it will be re-transmitted as implied in lines 7-11. Retransmission may result in duplicates. By line 13, however, only one copy of them will be kept in the buffer and fed to the game. In the event that the remote site or the network fails, the local site will be stuck in the loop freezing the game until it is recovered. It does not make more sense to allow the player to proceed alone.

3.2. Algorithm for Real-time Consistency

In Algorithm 1, the two functions, BeginFrameTiming() and EndFrameTiming(), work together to ensure real-time consistency. We first discuss a few concepts:

- **CFPS**: the constant number of frames that the game is expected to deliver in a second. The value of CFPS is game-specific but it is normally 60.
- **FPS**: frames per second, the actual speed of the game.
- **RTT**: the round-trip time between the two sites. The one-way network latency is estimated by RTT/2.

A naive way to control the game speed is to consume what is left in the current frame time by waiting. This works fine in a single-host system. Since the FPS is constant, the VM just needs to block until end of current frame time. However, in a network environment, naive waiting will not work because network latencies may cause a frame (more specifically, SyncInput) to take longer than 1/CFPS. In this case, the subsequent frames must compensate for the delay.

We devise Algorithms 3 and 4 for this purpose. They use the following four global variables:

- **AdjustTimeDelta**: the delay carried over from the preceding frame that the current frame should compensate for. It is initialized as zero.
- **TimePerFrame**: the expected time per frame, $\frac{1}{CFPS}$.
- **CurrFrameStart**: beginning time of current frame.
- **CurrFrameEnd**: the expected ending time of current frame, i.e., when it *should* end.

BeginFrameTiming() can be as simple as to call the OS to get the current system time and record it in variable CurrFrameStart. Then, as shown in Algorithm 3, EndFrameTiming() estimates the time when the current frame should end by adding the expected TimePerFrame and AdjustTimeDelta to CurrFrameStart. If the resulting CurrFrameEnd is smaller than the current system time, the current frame has taken longer time than it should. Then this lag should be compensated in the following frame by AdjustTimeDelta, which is a negative value. Otherwise, the current time has ended faster than expected. We set AdjustTimeDelta to zero and consume the remaining time of current frame by waiting.

In what has been discussed so far, algorithm EndFrameTiming() compensates for the latencies in SyncInput(). However, there is a situation that it does not handle well. Consider the case in which two sites cannot begin at exactly the same time, which is not uncommon in practice. Then, depending on the initialization time difference, the earlier site has to wait for the later site in SyncInput(). The waiting can cause the earlier site to lag behind. Then the lag is compensated in the next frames of the earlier site as result of algorithm EndFrameTiming() to speed them up. The

Algorithm 3 Pseudocode of EndFrameTiming()

```
1: CurrFrameEnd = CurrFrameStart
                + TimePerFrame + AdjustTimeDelta;
2: CurrTime = get_current_time();
3: if CurrFrameEnd < CurrTime then
4:    AdjustTimeDelta = CurrFrameEnd - CurrTime;
5: else
6:    AdjustTimeDelta = 0;
7:    Wait for time of (CurrFrameEnd - CurrTime);
8: end if
```

Algorithm 4 Pseudocode of BeginFrameTiming()

```
1: var Time SyncAdjustTimeDelta, CurrTime;
2: CurrFrameStart = CurrTime = get_current_time();
3: if MySiteNo = 0 then
4:    SyncAdjustTimeDelta = 0;
5: else if MySiteNo != 0 then
6:    MasterFrame = LastRcvFrame[0] - BufFrame;
7:    SyncAdjustTimeDelta =
          (Frame - MasterFrame) * TimePerFrame
          - (CurrTime - (MasterRcvTime - RTT/2)) ;
8: end if
9: AdjustTimeDelta += SyncAdjustTimeDelta;
```

speed-up again causes wait in SyncInput(), and so forth. In this way, the site that starts earlier is always penalized if the initialization lag cannot be smoothed out because of reasons such as network latencies. The earlier site will suffer from considerable speed fluctuation.

Hence we need to design algorithm BeginFrameTime() more carefully, as shown in Algorithm 4. The main idea is to smooth out deviations between the speeds of two sites based on the speed of one reference site. To achieve so, we distinguish two types of sites: a master site (let it be site 0 for simplicity) that provides the reference speed, and a slave site (site 1 in a two-site configuration) that adjusts its pace according to the reference speed. We introduce the following three notations:

- **MasterFrame**: The latest known frame of site 0, which is LastRcvFrame[0] - BufFrame by Algorithm 2 (line 1). This is because the received frame has already counted local lag.
- **MasterRcvTime**: The time when site 1 receives the partial input to frame LastRcvFrame[0] from site 0.
- **SlaveFrame**: The current frame of site 1, at which site 1 computes the speed deviation. It is actually the global variable *Frame* in Algorithm 1.

Based on the above notations, the slave site (site 1) estimates the time when the master sent the MasterFrame by t = MasterRcvTime - RTT/2. The time difference between current time (site 1's local time) and the time the master sent the MasterFrame is estimated by d = CurrTime - t. Then the number of frames elapsed during this period is estimated by fn = d/TimePerFrame. The following equation (4) estimates The frame of the master site at current time (of site 1) is estimated by f = MasterFrame + fn. The difference of frames between the two sites at current time (of site 1) is estimated by fd = SlaveFrame - f = Frame - f. If the slave site lags behind, the value will be negative; and if the slave is faster, the value will be positive. We estimate the time deviation between the sites by SyncAdjustTimeDelta = fd * TimePerFrame. It is easy to verify that this is how we compute SyncAdjustTimeDelta in Algorithm 4.

Then we add SyncAdjustTimeDelta to the aforementioned global variable AdjustTimeDelta. Note that in Algorithm 4 we only adjust the speed of the slave site according to the speed of the master site. In the master site, the variable SyncAdjustTimeDelta is always zero. As a result, no matter which site starts earlier, the slave site tries to compensate for the startup time deviation: if site 0 starts earlier, site 1 will catch up; if site 1 starts earlier, it will slow down to wait for site 0; in either case the two sites will be synchronized within a short period and stay in a synchronized state. No site will be penalized. This contrasts the result of not using Algorithm 4, in which the earlier site will suffer from speed fluctuation. In practice, the startup time deviation between two sites will not be overly large. In our system, a simple session control protocol is implemented to ensure that two sites start at almost the same time, with at most one round-trip time deviation. Then the slave site can smooth out the deviation within only a few frames.

4 Performance Evaluation

From the perspective of realtime multi-player gaming, we consider the following two metrics:

- **Smoothness**: how the game is paced at each site.
- **Synchrony**: how closely two sites are synchronized.

To evaluate smoothness and synchrony, we need to measure the frame time of one site as well as the deviation between the frame times of two sites. Therefore we conduct two series of experiments: Series 1 will measure the average frame time of one site under different network conditions to evaluate speed of the game, and the average deviation of frame time to see how smooth the game is. Series 2 will measure the time difference between two sites for the same frames to evaluate synchrony of the two sites. In addition, the results of series 1 and 2 wlll also verify the ideal network conditions for our system to be reasonably effective.

We use a network emulator, Netem[8], to emulate the Internet environment. It runs on a Gentoo linux (kernel

[8] http://www.linux-foundation.org/en/Net:Netem

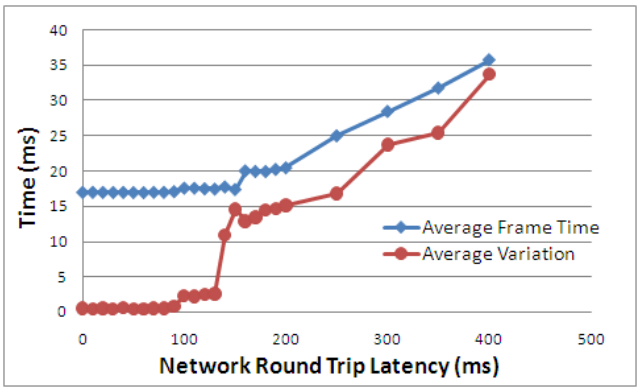

Figure 1. Frame rates and smoothness

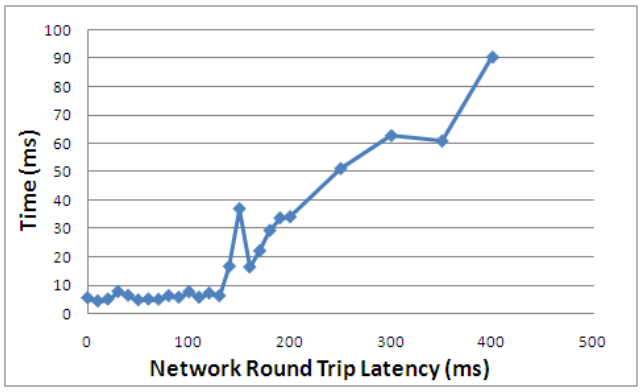

Figure 2. Synchrony between two sites

2.6.22-r9) server computer with one Intel Pentium IV 2.4 GHz CPU and 1 GB RAM. The Netem server has two network cards that are bridged, each being connected to one gaming client PC running MS Windows XP SP2. The two PCs are also connected similarly to a time server for measuring game times on the two PCs without having to synchronize their physical clocks. On the two PCs, our system is installed and the same game image of Street Fighter 2 is loaded. [9] The actual game does not affect the results.

4.1. Experimental Results

4.1.1. Experiment Series 1

To emulate a range of local and wide-area latencies, we experiment on round-trip times ranging from 0 to 400 milliseconds. Since the local lag is 100ms, the threshold RTT should not deviate too far away from 200ms. To see the threshold clearly, the step is set to 10ms from 0 to 200ms and 50ms from 200ms to 400ms. We do one experiment for each of those latency values. During each experiment, we record the beginning time of every frame of each site to the time server. The experiment stops after recording 3,600 frames. After that, we first calculate each frame time and then average times of the 3,600 frames for each experiment. Finally, we calculate the average of the absolute deviations of the frame times. [10] The results are plotted in Figure 1.

From Figure 1 we find that, when RTT is from 0 to 140ms, the average frame times are about 17ms which is the normal game speed of 60 FPS. The frame time increases as the network latency increases. As a result, the speed of the game gradually slows down. When RTT is from 0 to 90ms, the average deviation is almost 0ms. When RTT is from 100ms to 130ms, the average deviation is still under 5ms. However, when RTT reaches beyond 140ms, the average deviation suddenly jumps to 11ms and over. Hence we identify the threshold RTT as around 140ms.

From Figure 1 we also find that, when RTT is 150ms, the average deviation is an inflection point, i.e., it goes above that of RTT=140ms and that of RTT=160ms. This is because 150ms is around the threshold even though its average frame time is still around 17ms. As a result, it is not stable and the deviation is large. However, when network latency is increased to 160ms, the average frame time is slowed down to around 20ms, or 50 FPS. Then the system can tolerate higher latencies and the deviation becomes smaller. Nevertheless, the consequence is that the speed of the game is degraded to 50 FPS, which is considerably slower than the expected frame rate 60 FPS. Therefore, we do not recommend to run our system over networks with RTT latencies higher than 140ms.

4.1.2. Experiment Series 2

In this series, we measure the time difference of every frame between two sites. For similar reasons as in series 1, we set the RTT step as 10ms during 0-200ms and as 50ms from 200-400ms. We do one experiment for each of those RTT values. During each experiment, every site sends a packet to the time server when every frame begins and the time server records the receiving time. We can safely consider the RTT in a LAN as under 1ms. Each experiment stops after 3600 frames and we calculate the time difference between the two sites for every frame and then calculate the their absolute average. [11] The results are plotted as in Figure 2.

From Figure 2 we see that when RTT varies from 0 to 130ms, the average of absolute differences is less than 10ms. When it goes above 140ms, however, the average of absolute differences quickly goes up. For reasons similar to what has been discussed above, the threshold is around 140ms and 150ms is an inflection point.

[9] http://en.wikipedia.org/wiki/Street_Fighter_II

[10] For n numbers, $x_1, x_2, \ldots, x_n$, their average is $\bar{x} = (x_1 + x_2 + ... + x_n)/n$ and the average deviation is $(|x_1 - \bar{x}| + |x_2 - \bar{x}| + \ldots + |x_n - \bar{x}|)/n$.

[11] The absolute average of $x_1, x_2, \ldots, x_n$ is $(|x_1| + |x_2| + \ldots + |x_n|)/n$.

4.2. Discussions

There seems a discrepancy between the 100ms local lag and the 140ms experimental RTT threshold. In fact, 140ms RTT means 70ms one-way delay. Hence, the 100ms local lag should be considered as the maximum one-way delay that our system can tolerate. As shown in Figure 2, under the threshold condition (RTT=140ms), the average absolute synchrony deviation between the two sites is about 15ms. To prepare for deviations, this 15ms should be deducted from the 100ms maximum. In addition, each site sends one message every 20ms. In Algorithm 3, conceptually lines 7-11 may send messages at a higher rate. In the actual implementation, however, we need to strike a balance between interactivity and utilization of system resources (such as CPU and bandwidths). Hence all outbound messages are buffered. As a result, there may be an average delay of 10ms (worst-case 20ms) between the time an action is input and the time it is sent out. We should also take this 10ms away from the 100ms maximum. In addition, production and consumption of messages are performed by two threads. Assuming the thread time slice is 10ms, there is a 5ms average delay between the time a message is put in the buffer and the time it is sent. Therefore, the threshold one-way time should be around 100ms - 15ms - 10ms - 5ms = 70ms. This is why the RTT threshold is about 140ms.

We fix local lag at 100ms rather than adapt it to network conditions, esp. RTT. The reasons are explained as follows. As discussed above, when RTT is above 140ms, local lag will likely be above 100ms. No matter whether or not local lag is dynamically adapted to RTT, the user will experience noticeably degraded local responsiveness. Hence it does not pay off to implement adaptive local lag, which would add to system complexity. On the other hand, when RTT is below 140ms, fixing local lag at 100ms satisfies the interactivity requirements while keeping the implementation simple. A lowered local lag would increase the user's sensitivity to network conditions. The user would have to continuously adjust herself to thrashing game speed as the network speed fluctuates. Hence adapting local lag does not pay off, either.

5. Related Work

A large number of collaboration transparency systems have been developed, e.g., [1, 6, 7, 8, 14]. which adapt single-user software applications into multi-user applications without modifying their source code. Our work is the first that extends collaboration transparency techniques to the realtime gaming domain in which multiple players must be allowed to input simultaneously and the states must be synchronized in real time. By comparison, in traditional systems, simultaneous inputs are often disallowed [1, 6] or allowed only when application semantics is known [7, 8, 14]. Moreover, those systems only need to guarantee that different sites eventually see the same view [1, 6] or replicas of the shared data converge [7, 8, 14] without having to achieve view or data consistency in real time.

As in traditional collaboration transparency systems, the assumption underlying our work is determinism of the VM. In our work, we replicate the same game image and guarantee the same initial state as well as the same sequence of *user* inputs. The game VM from which we extended, MAME, is deterministic after careful analyses. Therefore, the general difficulties of synchronization in replicated collaboration transparency systems (as noted in [6]) are not found in our system. Nevertheless, accesses to host-dependent external resources such as system clocks, environment variables, and disk files may be sources of nondeterminisms since they provide other types of inputs (from the system instead of the user). In the case that accesses to those resources exist, they must also be coordinated as well to ensure same inputs to all VMs [6].

According to the classification in [9], our work is a type of replicated continuous application because its state changes are not only caused by users' actions but also the passage of time. Several techniques are widely used for consistency control in this domain, such as local lag, timewarp, and dead reckoning [5, 3, 9]. Timewarp needs to rollback application states, which may be used in realtime systems if the costs of rolling back are not too high. It is not applicable for solving our problem because rolling back states of a distributed game without semantic knowledge can be expensive and disrupt the game players' experience. Dead reckoning, on the other hand, needs to know the semantics of the applications in order to make informed predictions. It cannot be used in our context because our system does not assume knowledge about the games to avoid being bound to specific games. Our work uses local lag to delay the execution of local actions until remote actions on the same frame are received. Our contribution is to extend a well-known technique for solving a new problem, i.e., real-time consistency control without modifying source code or assuming semantic knowledge of the games.

In addition, our work addresses the real-time constraints at the application level. This way we avoid making assumptions about availability of QoS support at lower levels, such as RSVP [15] and DiffServ [2]. Also at the application level, many distributed multimedia systems explore how to satisfy the real-time constraints by varying the amount of transmitted data under different network conditions (e.g., the MPEG standards). However, those techniques are not applicable in our system. As discussed in Section 3, consistency in our work relies on reliable transmission of input data; the amount of data is not excessive; reducing or discarding input data would lead to logical inconsistencies; and user inputs cannot be known ahead of time to be buffered.

6. Conclusions

This paper presents an approach to adapting a single-machine game VM into a distributed gaming system. By this approach, a huge number of legacy games emulated by the original VM are turned into distributed games without any modification. As a main contribution, it initiates a new paradigm of so-called real-time collaboration transparency in a new application area, multi-player network gaming. At the heart of this approach is a novel synchronization algorithm. The main challenge in this context is how to implement both logical and real-time consistency control without assuming semantic knowledge of the games. This work extends a well-known method, local lag, to a new application domain that has not been explored to the best of our knowledge. We have implemented the system and evaluated its performance under a variety of network conditions. Our experimental results show that our system works the best when network round-trip delay is under around 140ms. As suggested in the analyses, any system taking a similar approach is subject to a similar latency constraint.

The journal version of this paper [16] addresses issues such as how to support multiple players and observers, how to accommodate late comers, and how the system performs in presence of packet losses. In future research, we plan to extend this work on mobile devices.

7. Acknowledgement

The authors thank the anonymous reviewers for their insightful feedback and Liping Gao and other members of the Cooperative Information and Systems Laboratory at Fudan University for their great support. The work is supported by National Natural Science Foundation of China (NSFC) under Grant No. 60736020, National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321905, Shanghai Science & Technology Committee Key Fundamental Research Project under Grant No. 08JC1402700 and Shanghai Leading Academic Discipline Project under Grant No. B114.

References

[1] H. Abdel-Wahab and M. Feit. XTV: a framework for sharing X window clients in remote synchronous collaboration. In *Proceedings of IEEE TriComm'91 Conference on Communications for Distributed Applications and Systems*, pages 159–167, Apr. 1991.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated service. Network Working Group Request for Comments: 2475, http://www.ietf.org/rfc/rfc2475.txt, 1998.

[3] S. Ferretti and M. Roccetti. Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games. In *Proceedings of the ACM SIGCHI Conference on Advances in Computer Entertainment Technologies*, pages 405–412, 2005.

[4] G. C. H. Chen, L. Chen. Effects of local-lag mechanism on task performance in a desktop cve system. *Journal of Computer Science and Technology*, 20(3):396–401, 2005.

[5] Y. Ishibashi, Y. Hashimoto, T. Ikedo, and S. Sugawara. Adaptive causality control with adaptive dead-reckoning in networked games. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 75–80, New York, NY, USA, 2007. ACM.

[6] J. C. Lauwers, T. A. Joseph, K. A. Lantz, and A. L. Romanow. Replicated architectures for shared window systems: a critique. In *Proceedings of the ACM SIGOIS Conference on Office Information Systems*, pages 249–260, 1990.

[7] D. Li and R. Li. Transparent sharing and interoperation of heterogeneous single-user applications. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 246–255, Nov. 2002.

[8] D. Li and J. Lu. A lightweight approach to transparent sharing of familiar single-user editors. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 139–148, Nov. 2006.

[9] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, Feb. 2004.

[10] L. Pantel and L. C. Wolf. On the impact of delay on real-time multiplayer games. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29, New York, NY, USA, 2002. ACM.

[11] K. S. Park and R. Kenyon. Effects of network characteristics on human performance in a collaborative virtual environment. *Proceedings of the IEEE Virtual Reality Conference*, pages 104–111, 1999.

[12] J. D. Pellegrino and C. Dovrolis. Bandwidth requirement and state consistency in three multiplayer game architectures. In *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 52–59, New York, NY, USA, 2003. ACM.

[13] B. Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Survey*, 16(3):265–285, 1984.

[14] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction*, 13(4):531–582, Dec. 2006.

[15] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *Readings in multimedia computing and networking*, pages 624–634, 2001.

[16] S. Zhao, D. Li, H. Gu, B. Shao, and N. Gu. A middleware approach to converting legacy TV/aracde games for real-time distributed gaming. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2009. Under review.