Time critical synchronization of networked rendering

Jenny Olsson

Stockholm 2013
KTH

Abstract

This bachelor thesis explores the possibility of synchronizing rendering over a network.

One possible solution for achieving synchronization is presented in the form of a demonstration program written in Python. This solution combines several different known solutions and algorithms. The design choices for the demonstration program is explained and evaluated as well as the chosen algorithms.

Summary

This thesis was written in the spring of 2013. It is an investigation into different solutions on how to sychronize rendering on different computers over a network. A possible use case is to manage animations that need to combine a number of monitors to achieve a desired resolution, since monitors have a limited area. The specifications require the possibility to distribute events in real time to all rendering parts, for example by turning a knob on a mixer-table. The event message should contain parameters for the animation that the rendering parts should render. The rendering of these events should also be synchronized.

Different solutions on how to achieve synchronization have been investigated. A possible solution written in Python is presented. This solution is implemented using a combination of well known algorithms and solutions.

Preface

This thesis was written at 23c in Stockholm as the completion of my bachelors degree in Computer Engineering from The Royal Institute of Technology in Kista. It was started in March of 2013 and consists of 15 ECTS points.

It presents a possible solution on how to synchronize the rendering processes on different computers over a network.

This thesis and the code presented has been written by me and in the cases where other sources has been used this is clearly cited.

Stockholm 2013

Jenny Olsson

Glossary

NTP

Network Time Protocol, used to synchronize computer clocks over a network.

Frameskipping

Skipping forward a certain number of frames in an animation.

Tweening

Also known as inbetweening. Technique for creating linear animations by defining the start and the end states of the animation.

Timestep

Defining the amount of time between two steps in the animation, making the animation depend on time rather than framerate.

Round trip delay

The delay in time between sending a message and recieving a reply.

Acknowledgements

I would like to thank 23 Critters, especially Mathias Tervo and Kristoffer Smedlund, for presenting me with the problem and having great patience in discussing it with me. I would also like to thank Minna Morri, the worlds best lab partner, for her support, not only in the making of this thesis.

Finally I would like to thank Sanna Barsk for the extensive proof-reading of this thesis.

Contents

Chapter 1

About synchronization

You may delay, but time will not.

- Benjamin Franklin

1.1 Background

Because projectors and screens have a limited resolution, achieving a specific higher resolution requires combining several screens or projectors. This leads to a need to synchronize the graphics-rendering of an arbitrary number of computers, connected to these screens or projectors. This thesis will explore the possibility of synchronizing rendering.

The work of this thesis will ultimately be used in a system to synchronize the rendering in a platform for graphics visualisation written in C++, OpenGL3 and GLSL. According to the specification, events also need to be able to be sent to the clients, and the clients should in turn be able to perform specific animations on recieving these events.

1.1.1 Rendering graphics

Computer graphics are rendered at the speed of the framerate. Framerate is the speed in which the the image is repainted, measured in frames per second, FPS.

A commonly used practice when rendering graphics is to have a function named draw. This function will draw one image and is called repeatedly in the draw loop. Its common to animate an object by using the framecount, a counter that is incremented by one for each frame. This will tie the speed of the animation to the framerate. An example of this is shown in pseudocode below.

```
pos.y = height * sin(framecount)
pos.x = width * cos(framecount)
```

To decouple animation speed from framerate the animations can be made timedependent. The draw function will then take a timestamp as input and derive animations from this timestamp, making sure that the animation is in the same state regardless of the framecount. An example of this is shown below in pseudocode.

```
pos.y = height * sin(timestamp)
pos.x = width * cos(timestamp)
```

1.1.2 When is the rendering synchronized?

Research has been made into the maximum amount of delay acceptable for the human eye between audio and video[1], as this is important in the broadcasting of television. Research into what could be "acceptable" asychronization for the human eye in video to video synchronization is outside the scope of this thesis, it will be assumed that it is a soft real time system and that the delay between the video should be as short as possible, a best effort system. For the sake of clarity we discuss delays between animations in milliseconds.

The the acceptable amount of delay between interaction and animation has been set to 50 ms in the application.

[1]http://tech.ebu.ch/docs/r/r037.pdf, EBU

1.1.3 What needs to be synchronized?

Since the clients rendering the animations will be distributed on different computers it cannot be assumed that the clocks on these computers are in sync with each other. This means that the computer clock time cannot be used directly for deciding the time of the animations. It also cannot be assumed that the computers rendering the animations are running at the same framerate.

Another part of the problem is that the clients will not recieve the event messages from the server at the same time since they are distributed over a network and can have very different delays. The animations that occur on recieving events therefore also needs to be synchronized. This could be solved by giving the clients some notion of their delay in relation to the other clients.

1.2 Synchronizing clocks over a network

The problem of synchronizing computer clocks over a network has been investigated since the early days of networks. There are a few well known algorithms, these are the ones looked into in the making of this thesis.

1.2.1 The NTP algorithm

The NTP protocol[3] was originally devlopeded in 1985 by David L.Mills. It is used to synchronize clocks over a network by calculating the master offset using the round trip delay. NTP is still in use today, the latest RTC is from June 2010[2].

The NTP algorithm uses four different timestamps to calculate the round trip delay.

t0 is the time of the request packet transmission

t1 is the time of the request packet reception

t2 is the time of the response packet transmission

t3 is the time of the response packet reception.

[2] https://tools.ietf.org/html/rfc5905

The timestamps of t0 and t3 is set by the sender and t1 and t2 by the reciever. The sender then calculates the round trip delay, δ.

$$\delta = (\mathrm{t3} - \mathrm{t0}) - (\mathrm{t2} - \mathrm{t1})$$

NTP assumes that the delay of the sender and the reciever is equal, and thus calculates the master offset, θ, as below.

$$\theta = \frac{(\mathrm{t3} - \mathrm{t0}) - (\mathrm{t2} - \mathrm{t1})}{2}$$

1.2.2 The Berkley algorithm

The Berkley algorithm was written by Gusella and Zatti in 1989. A simplification of the steps of the algorithm is shown below.

1. A master polls slaves, the slaves reply with their time.
2. The master uses the round-trip time of the messages to estimate the time of each slave and the master's own time.
3. The master calculates an average of the clock times, ignoring extreme values.
4. The master sends the slaves their delta, which can be positive or negative.

The delta value is used by each slave to adapt their time to the choosen master time.

1.2.3 Cristians algorithm

Cristians algorithm[1], developed in 1989 by Flaviu Cristian, is used for synchronizing clocks by calculating the round trip time. It is a probabilistic algorithm[1], meaning that it it delivers a better accuracy the shorter the round trip delay time is.

The algorithm works as follows. The server sends a message to the client, and the client replies. When the server recieves the reply from the client it calculates the round trip delay time, that is, the time passed between sending the request from the server and recieving the reply from the client divided by 2. This assumes that the latency of the server and the client is equal.

The server then sends the sum of its own time added with the round trip delay time to the client, and the client sets this as its own time.

$$\text{Client time} = \text{Server time} + \text{Round trip delay time}$$

Figure 1.1: Setting the time on the Client

1.2.4 PTP and GPS

The Precision Time Protocol, abbriviates PTP, is a protocol used for high accuracy synchronization in time critical systems. The first RTC is from 2002[4]. PTP achieves a very high level of synchronization but it was assumed in the making of this thesis that this kind of precision would not be needed for this purpose.

One additional solution for synchronizing the time of distributed applications is to use GPS. Since this would require that all clients had a GPS-sender/reciever, something that seemed improbable, GPS was not further investigated as a solution in this thesis.

1.3 Problem definition

- Can we synchronize the rendering on an arbitrary number of computers?
  - How can we solve the messaging of synchronization-events?
  - When and how often do we need to synchronize?
  - What techniques are available?
  - Which method gives the highest level of synchronization?
  - Which method is the most efficient?
  - How can the synchronization be optimized?

Chapter 2

Design of the application

This section aims to explain the architecture and design choices made for the demonstration application.

The code for the program described is available at github[1].

2.1 Overall architecture

The demonstration program consists of a server that continously accepts connecting clients and a client that connects with the server. An arbitrary number of clients can run and connect to the server at the same time.

2.2 Why the Client-Server approach?

The client-server approach was chosen, as opposed to choosing a master among the clients, since there was a need to both generate and distribute specific events from a central source.

[1] https://github.com/Rymdsnigel/thesis-demo

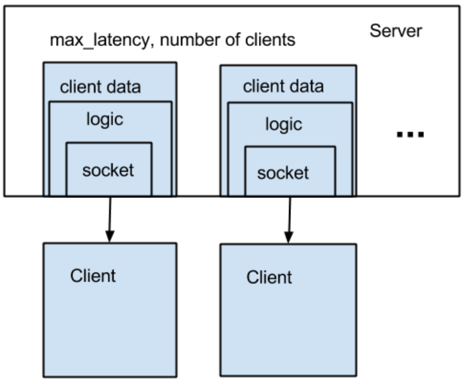


Figure 2.1: Overall architecture of the system.

2.3 Events

Both the client and the server create events that will be parsed to JSON and sent either from the server to the clients or from the client to the server. The server produces render events on input from the pygame window, sync events on key input from the pygame window and latency update events as a reply to sych events from the client, while the client only produces sync events as reply to the server sync events. All three different types of events have an event type, an integer that identifies the type of event.

The content of the sync event and the latency update event will be explained in chapter 4. The three different types of events are shown below.

latency_update_event	
"event_type"	0
"latency"	latency
"max_latency"	max latency

sync_event	
"event_type"	1
"recieved_at"	recieved at
"sent_at"	sent at
"delta"	delta
"client_id"	client id

render_event	
"event_type"	2
"id"	id
"channel"	channel
"data_id"	data id
"data_val"	data val
"timestamp"	timestamp
"reserved"	reserved

2.4 Event queues

Both the server and the clients have event queues, when they create an event they place it in their queue. Messages are then polled from the queue and sent one at the time. This works in the same way on the clients and the server and the purpose is to avoid blocking other threads by sending data.

2.5 Messaging

The server and the clients send and recieve JSON, the json is created using simplejson library functions, dumps() for creating JSON from a dict and loads() for unpacking the dict from JSON. The functions for generating the dicts that will be sent as the messages are specified in event.py, and the specified dicts are shown in the section about events [2].

The choice of JSON for messaging, rather than using pickle, cpickle or XML was made due to JSONs speed of reading and writing [3] and to support future flexibility in language since pickle and cpickle is python-specific.

[2] 2.3
[3] http://kovshenin.com/2010/pickle-vs-json-which-is-faster/, Kovshenin

2.6 Communication

2.6.1 Protocols

The choice of TCP-sockets as opposed to UDP-sockets was made at an early stage of development. Although UDP is typically the ideal choice for time critical applications, customizing the needed control mechanisms would be outside the scope of this thesis.

The choice of TCP presented proved to be problematic when it was discovered that TCP:s message buffering, using Nagels algoritm, generated a general delay in messaging, a delay of 20 ms both from the server as from the clients. This was discovered by measuring the delays of the application over localhost, where network delays should be close to 0. This issue also resulted in that more than one json object could be put on the queue of recieved events, which lead to a json-parsing error when trying to load the objects. These errors and the buffer delays were removed by disabling Nagels algorithm[4] by setting the nodelay flag on the socket.

```
self.s.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
```

Figure 2.2: Setting the nodelay flag on the socket

2.6.2 Sockets

The demo server communicates with the clients via gevent sockets since the sockets need to be threaded in order to not block the other processes.

2.6.3 Replacing the network layer

Functional cohesion has been strived for, in order to make the part of the code pertaining to transport easily replaced by for example an implementation using UDP or implementation of a ready solution such as redis.

[4] http://stackoverflow.com/questions/8617809/unstable-tcp-receive-times

2.7 Threading

Both the server and the client have to achieve concurrency. This is done by letting both the TransportServer and the TransportClient inherit from gevent Greenlets. Greenlets are pseudothreads that share the same OS-thread, and cooperatively multitask. Because it is cooperative, the threads must release control of critical operations to avoid blocking other threads.

2.8 Animations

For animations as well as server input Pygame was chosen, because of its simplicity to work with. Pygame is built on SDL[5] and is a Python library for game development.

2.8.1 Tweening

Since it is necessary to be able to manipulate the time when every client performs a specific animation one of the first steps was to make the animations time dependent and independent from framerate. This is solved in the Tween class. The Tween class has two functions for generating input for the animations. This generated input can be used for, for example, deciding the position of an object or the color of a part of an object.

In every frame drawn on the client the current time is sent to the animation's step function in the client's Tween instance. The animation saves the current time between frames and can then calculate the delay between two frames (delta). The animation then uses the delta value to interpolate between the start and the end values, ensuring that the animation runs for a set amount of time. This way the time of the animation can be manipulated by giving the Tween functions a value for current time that is in sync with the other clients.

Making the animations time dependant by timestepping is a major part of the chosen solution for synchronizing the animations presented in this thesis.

The functions for generating data for the animations are in the Tween class, see appendix B.

[5] http://www.pygame.org/wiki/about

Chapter 3

Synchronizing the clients

The synchronization of the animations has two major steps. First of all there is a continously running animation that takes input from the Tween class. This animation can be synchronized by giving every client a notion of how much delay there is between that client and a master, a master which could be the server.

The second step is to adapt the time the clients play specific parts of the animation after recieving a render event message from the server. An optimization of this is to frameskip the clients that are slower than a specified threshold.

3.1 Synchronizing the time

Because the clients may run on different computers the computer clock time cannot be used as a parameter to the animations since the computer clocks are unlikely to be in sync with each other. As shown in chapter 1 this is a well known problem in distributed systems, so the first step would be to implement one or several existing synchronization algorithms.

A combination of the Berkley algoritm described in 1.2.2 and NTP described in 1.2.1 was used to synchronize the time of the clients and the server. Both

NTP and the Berkley algorithm are designed to be used in intranets since they assume an evenly distributed delay. The end product that the work in this thesis is intended to be used in is assumed to run on an intranet. The accuracy given by these algoritms is assumed to be good enough.

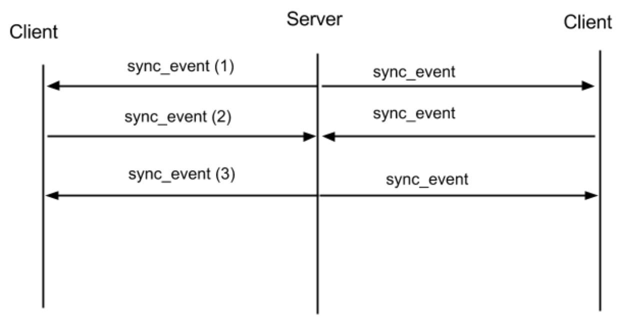
An artificial time is created both on the server and on the clients on start and this time is never manipulated. Instead a delta value added to the artificial time is used as input to the animations.

3.1.1 Distributing deltas

The server initiates the synchronization with the clients by sending a sync event message to the clients. The clients reply with the time they recieved the message, the time they send their reply and their old delta-value. The server then calculates each clients delta and sends each client a message containing their new delta.

If the value of the old delta of the client and the new delta calculated by the server differs more than 2 milliseconds the server will send a new sync event as a message to the client, repeating the procedure until the delta is stable.

Every time a new client connects to the server this client needs to synchronize its time with the server's.



1: Message contains client_id
2: Message contains two timestamps, recieved at and sent at.
3: Message contains clients delta.

3.1.2 Calculating deltas

An implementation of NTP was used to calculate each clients latency, shown below. This calculation is done on the server and the delta value calculated is the NTP master offset.

```
self.delta = ((self.t_1 - self.t_0) + (self.t_2 - self.t_3))/2
```

3.1.3 Using the delta values

All animations must take a timestamp as a parameter for playback. This is used to timestep through the animation. To create the timestamp the client uses the time acquired when the delta is added to its own local time. This way all client's animations will be synchronized, they will be at the same stage in the animation at the same time given that they have their correct delta value.

3.2 Latencies

3.2.1 Calculating the latency of a client

The network latency of a client is calculated in the same function on the server as the delta value is calculated. This way the values of t0-t3 can be reused for calculating the latency.

The latency is assumed to be evenly distributed and is calculated as shown below.

```
self.latency = (self.t_3 - self.t_0)/2
```

The latency of each client is calculated by the server and sent to the client in a latency_update_event, along with the maximum latency. The maximum latency is the latency of the client with the greatest latency. The client then calculates the wait period used to synchronize the client to the client with the highest latency by subtracting its own latency from the maximum latency.

Client latencies are stored in an array on the server. Every time a client latency is updated the value of that respective row in the array is updated. The maximum latency is simply the largest number in this array. A previous attempt used just an integer variable for storing the latency, resetting it every time a larger latency was found. This introduced the risk of the maximum latency at some point potentially being set to a very high level, not reflecting the current latencies.

Every time a new client connects to the server the server must send latency_update_events to all clients since the new connecting clients latency might be bigger than the current maximum. The maximum latency then needs to be redistributed to all clients.

$$\text{applied\_latency} = \text{maximum\_latency} - \text{latency}$$

Figure 3.1: Calculating the applied latency

3.2.2 Delaying animation start based on latency

Before handling a new render event the client waits for the number of milliseconds specified by its applied latency. This way the faster clients will compensate for the slower ones.

3.2.3 Skipping frames if delay is too long

If a clients latency is higher than a set threshold, the highest latency under the treshold is selected as the maximum latency and the client(s) above the threshold skip ahead instead. The clients that skip ahead will use their latency to skip that amount of time ahead in the animation.

This is done by giving the clients animation a value for current time with the value of the clients latency subtracted. This way the animation beleaves it is in a later stage of the animation, a stage matching the other clients animations.

Chapter 4

Running the demonstration application

This section explains how to run the demonstration application and how to interact with it.

4.1 Running the programs

The demo program is written in python 2.7.2. It requires pygame 1.9.1, simple-json 2.1.6, docopt 0.6.1 and gevent 0.13.0.

It is delivered with a Bash-script named testrun_2clients.sh that starts the server and two clients.

The server and clients can also be started separately, but the server must be started first since the clients have no autodiscovery. The server requires no parameters and can be started as shown below.

```
$ python server.py
there is no soundcard
```

Starting the clients requires some flags to be specified. The –help flag displays the flags the client takes as arguments on startup.

```
$ python client.py --help
Client rendering graphics

Usage:
  client.py [--port=<nr>]
            [--framerate=<frame/s>]
            [--x=<pixels>]
            [--y=<pixels>]
            [--pos <x1> <y1> <x2> <y2>]
  client.py (-h | --help)
  client.py --version

Options:
  -h --help     Show this screen.
  --version     Show version.
  --port=<nr>   Port number to bind to client [default: 5007].
  --framerate=<frame/s> Client framerate [default: 0].
  --x=<pixels> Width of client screen [default: 300].
  --y=<pixels> Height of client screen [default: 300].
  --pos <x1> <y1> <x2> <y2> Position of the part of the animation the client shows.
```

4.2 Emulating network delays

In the beginning of the development the emulated network-delays were inserted using an gevent.sleep for a variable time in the code. This variable could be specified from command line. This was replaced by using netem [1], a more flexible solution.

With netem, delays can be bound to specific ports. The current version therefore requires the clients to be bound to specific ports in order to test them on ports with preset delays. The port to bind the client to must be specified when starting the client, using the –port flag.

[1] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

```
[]
tc qdisc add dev lo handle 1: root htb

tc class add dev lo parent 1: classid 1:1 htb rate 1000Mbps

tc class add dev lo parent 1:1 classid 1:11 htb rate 100Mbps
tc class add dev lo parent 1:1 classid 1:12 htb rate 100Mbps
tc class add dev lo parent 1:1 classid 1:13 htb rate 100Mbps

tc qdisc add dev lo parent 1:11 handle 10: netem delay 40ms
tc qdisc add dev lo parent 1:12 handle 20: netem delay 20ms
tc qdisc add dev lo parent 1:13 handle 30: netem delay 0ms

tc filter add dev lo protocol ip prio 1 u32 match ip dport 10001 0xffff flowid 1:
tc filter add dev lo protocol ip prio 1 u32 match ip dport 10002 0xffff flowid 1:
tc filter add dev lo protocol ip prio 1 u32 match ip dport 10003 0xffff flowid 1:

tc filter add dev lo protocol ip prio 1 u32 match ip sport 10001 0xffff flowid 1:
tc filter add dev lo protocol ip prio 1 u32 match ip sport 10002 0xffff flowid 1:
tc filter add dev lo protocol ip prio 1 u32 match ip sport 10003 0xffff flowid 1:
```

Figure 4.1: Setting delays on port 10001, 10002 and 10003

```
tc qdisc del dev lo root
```

Figure 4.2: Removing delays set on dev

4.3 Interacting with the server

The pygame window of the server captures the events. When clicked at it will display an animation that changes the color of a cube and changes it back. By holding down the right mouse button the cube can be moved. By pressing any keyboard key a sync event is sent from the server to all clients.

4.4 Communication

The server continously accepts new connections but if the clients lose their connection to the server they will not automatically reconnect. This means that if the server should crash, the clients lose their connection to the server and the connection will not be reestablished by restarting the server.

CHAPTER 5

Conclusions

The goal of this thesis was to implement an algorithm for synchronizing rendering on different computers over a network. This has been achieved combining well known algorithms for synchronization and adapting the rendering to be time dependent.

Some important steps in the development were:

1. Making the animations on the clients depend on the time.
2. Making sure that the client and the server agreed on a timestamp.
3. Using the agreed time in the animation.
4. Calculating the network latency of a specific client.
5. Using the network latency to delay animating.
6. Finding a convenient way to emulate network latencys.

5.1 Testing the application

The application has during development been tested over localhost, running the server and all clients on the same computer, applying delays using netem as descibed in section 4.2. It has also been run on a local network with the server running on one computer and the clients on other computers.

The synchronization has been evaluated by logging the time when each client plays a specific step in the animation. Logs are written to file for evaluation. An example of a log that shows delays of around 30 ms before applying the latencies can be found in appendix A. The client ports in this testrun have delays of 20 ms respectively 50 ms which makes a 30 ms async seem reasonable. After applying the latencies we get a time gap between the animations of between 0 and 3 ms. Other testruns of the programs show delays between 0 and 8 ms after synchronizing.

Since the application is a distributed system, testing through logging of timestamps requires the clocks of the computers running the clients to be synchronized to make the logs useful. This is why the synchronization has only been investigated with all parts of the application running on one computer.

5.2 Limitations

Outlook – suggested improvements

The demo is a proof of concept and there are a few recommended improvements to it.

The first suggested step would be to evaluate the system using UDP or by replacing the network-layer with some solution like zeromq or redis. Both of these provides solutions for a publish/subscribe server.

The input for the server should be generalized so that the input device could be replaced by for example a mixer-table.

Appendix A

Log from two clients running on the same computer

```
2013-05-09 22:59:00,340 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:00,374 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:00,540 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:00,573 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:01,115 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:01,146 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:01,320 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:01,346 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:01,956 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:01,986 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:02,156 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:02,186 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:02,741 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:02,772 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:02,941 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:02,973 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:03,784 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:03,826 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:03,984 - INFO - 1- Client animation started, time for animation:
2013-05-09 22:59:04,026 - INFO - 2- Client animation started, time for animation:
2013-05-09 22:59:05,120 - INFO - 1- Client animation started, time for animation:
```

```
2013-05-09 22:59:05,159 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:05,321 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:05,358 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:05,932 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:05,964 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:06,132 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:06,166 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:11,041 - INFO - 1- Client applied latency: 29
2013-05-09 22:59:11,132 - INFO - 2- Client applied latency: 0
2013-05-09 22:59:12,464 - INFO - 1- Client applied latency: 30
2013-05-09 22:59:12,549 - INFO - 2- Client applied latency: 0
2013-05-09 22:59:13,751 - INFO - 1- Client applied latency: 29
2013-05-09 22:59:13,842 - INFO - 2- Client applied latency: 0
2013-05-09 22:59:15,048 - INFO - 1- Client applied latency: 29
2013-05-09 22:59:15,138 - INFO - 2- Client applied latency: 0
2013-05-09 22:59:16,771 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:16,772 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:16,971 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:16,971 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:17,614 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:17,615 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:17,814 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:17,815 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:18,431 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:18,432 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:18,631 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:18,631 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:19,271 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:19,272 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:19,471 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:19,473 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:20,176 - INFO - 1- Client animation started, time for animation: 20C
2013-05-09 22:59:20,176 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:20,375 - INFO - 2- Client animation started, time for animation: 20C
2013-05-09 22:59:20,376 - INFO - 1- Client animation started, time for animation: 20C
```

Appendix B

The Tween class

```
class Tween(object):

    def __init__(self, *args, **kwargs):
        self.running = False
        self.logger = logging.getLogger('client')


    def play(self, start, end, time=1000.0, reverse=False,
interpolation=None,timestamp=None, skip=0):
        self.start = start
        self.end = end
        self.delta = end-start
        self.time = time
        self.timestep = self.delta /self.time
        self.value = start
        self.running = False
        self.reverse = reverse
        self.increasing = (end>start)
        self.running = True
        self.skip = skip
        self.timestamp = timestamp - skip
# this will result in frameskipping if skip is not 0
```

```
        self.logger.info("Client animation started, time for animation:
" + str(self.time))

    def step(self, current_time):
        if self.running :
            diff = current_time - self.timestamp
            self.value += diff * self.timestep
            if (self.value >= self.end) == self.increasing:
                if self.reverse:
                    self.play(self.value, self.start, timestamp=current_time,
time=self.time, reverse=False)
                else:
                    self.running = False
                    if self.value > self.end:
                        self.value = self.end
        self.timestamp = current_time
```

Bibliography

[1] Flaviu Cristian. "Probabilistic Clock Synchronization." *Distributed Computing*, 3, page 146-158, 1989

[2] Andrew S. Tanenbaum, Maarten van Steen Distributed Systems, Principles and Paradigms, second edition Chapter *Synchronization*, pages 231-273, 2007.

[3] David L. Mills NTP Architecture, Protocol and Algorithms `http://www.eecis.udel.edu/~mills/database/brief/arch/arch.pdf` [Online; accessed 7th April 2013].

[4] 1588-2002 "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems" *2002*