# Reducing lag in a distributed physics system through ahead-of-time simulation

## An initial implementation

**Timothy Zonnenberg**[1]
**Supervisor(s): Ricardo Marroquim**[1]**, Amir Zaidi**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Student: Timothy Zonnenberg
Final project course: CSE3000 Research Project
Thesis committee: Ricardo Marroquim, Amir Zaidi, Chirag Raman

**Abstract**

Lag in Virtual Reality can be devastating to the enjoyment of participants. This paper reduces the lag in a physics system by implementing a subset of the architecture proposed by Loren Peitso and Don Brutzman[2]. This subset contains an event based physics system together with a networking system. The collision system of the physics system has been heavily reduced in size, as to speedup development of the architecture.

This architecture seems promising and handles predictable cases with almost no noticeable lag. This architecture does not handle external events well, as the lag is noticeable. The 'healing' process for unexpected external events should be researched further to make this architecture viable.

# 1 Introduction

Lag in any multiplayer environment can be frustrating. This is significantly worsened in Virtual Reality (VR) applications because participants expect that everything will work like it does in the real world. For example, participants will expect an instant response from the application. This lag leads to discomfort and a less appealing experience for participants.

Before going too much into the topic, it is important to define some of the terminology that will be used in this research paper. Latency is the actual measured delay between sending and receiving a specific piece of information. For this work, latency will refer exclusively to network latency. Lag on the other hand is how participants perceive the latency.

Latency could be optimized but can be considered a variable constant for the developer. The developer cannot change how the participant is connected to the internet, but they can change how their program handles this network latency.

The common approach to resolve lag is to mask it. This can be done by using lag compensation techniques[1]. These techniques are usually quite complex but are needed for an instant input response. The time between getting the new state from the server once the participant has given an input is significantly increased due to network latency. Thus, client-side prediction or other lag prevention techniques are required.

This work aims at implementing an architecture proposed by Loren Peitso and Don Brutzman[2], which tries to hide the latency in a physics system. This would result in the participants not experiencing any lag, even though latency is still present.

The architecture is an event-based architecture where each update of an object, collision check, and outside input happens in events. All the events have a specific time for when they need to be executed. Some events also schedule other events. There are also events that have some specific constraints, like the fact that for each physics object there can only exist one startMove event. For a full explanation of this architecture, we refer the reader to the original paper[2].

# 2 Background

## 2.1 Multiplayer Game Development

Normally a multiplayer game uses an authoritative server, where the server has full authority over the game state. A naive implementation of this system sends the participants desired input to the server, computes the resulting state on the server and send this state back to the user. Once this information arrives on the client, the client starts moving or reacting to

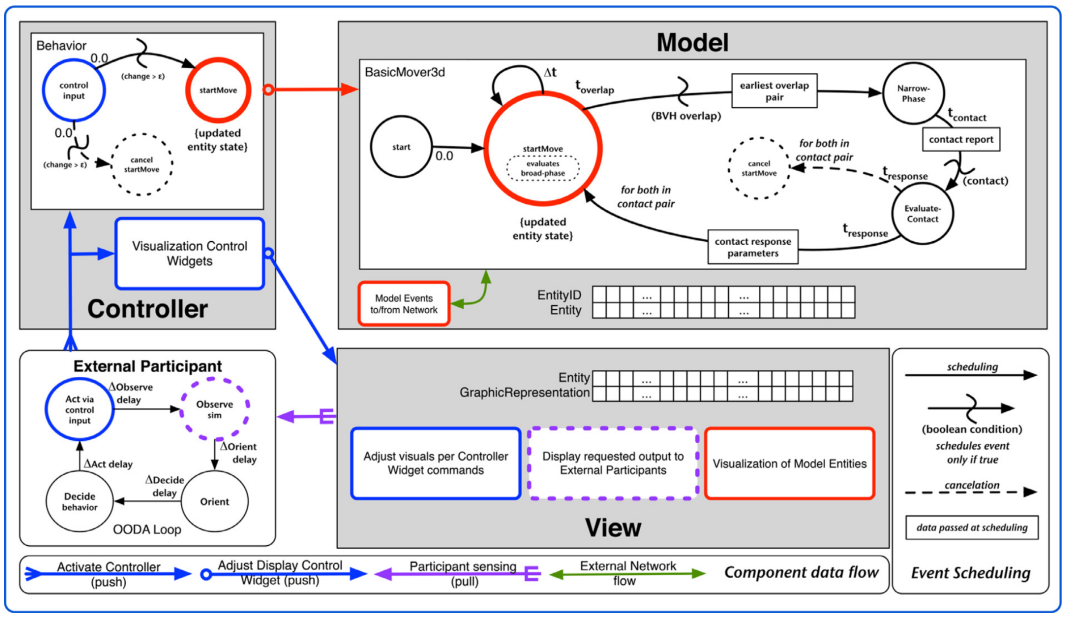


**Figure 1:** The architecture discussed in the paper[2]

the system. This has a negative impact for the user experience since they expect an instant reaction. To reduce this lag, client-side prediction[1] needs to be implemented. With client side prediction, the client starts moving on their side at the same time the user input is given. Once the information from the server has arrived, the client checks if the response matches the predicted state. The client then either continues, interpolates, or teleports the player state, depending on the amount of error. The exact details of how to handle these cases depens on the game.

## 2.2 Physics simulations

Numerical methods[3] are key in physics simulations. In short, numerical methods solve ordinary differential equations (ODEs). In a physics simulation numerical methods are used to calculate the acceleration of objects for that specific frame. To calculate the acceleration, Newton's second law of motion is used. Once the acceleration is known, it can be used to calculate the other necessary variables. To derive the function of velocity, the definition of acceleration is used. Likewise, the definition of velocity is used to calculate the position of the physics object. In summary, this results in the following equations:

$$\mathbf{F} = m\mathbf{a} \quad \text{(Newton's second law of motion)} \tag{1}$$

$$\mathbf{a} = \Delta\mathbf{v}/\Delta t \quad \text{(Definition of acceleration)} \tag{2}$$

$$\mathbf{v} = \Delta\mathbf{s}/\Delta t \quad \text{(Definition of velocity)} \tag{3}$$

Where $\mathbf{F}$ is the total force applied to the object, $\mathbf{a}$ is the acceleration, $\mathbf{v}$ is the velocity, $\mathbf{s}$ is the distance, and $t$ is the time.

These functions need to be transformed into a discretized form that can easily be implemented and used by simulations.

$$\mathbf{a} = \mathbf{F}/m \tag{4}$$

$$\Delta\mathbf{v} = \mathbf{a} * \Delta t \rightarrow \mathbf{v}_n = \mathbf{v}_{n-1} + \mathbf{a} * \Delta t \tag{5}$$

$$\Delta\mathbf{s} = \mathbf{v} * \Delta t \rightarrow \mathbf{s}_n = \mathbf{s}_{n-1} + \mathbf{v} * \Delta t \tag{6}$$

A numerical method can be used to approximate a function which cannot be solved analytically. The problem for an analytical solution is the recursive dependency between function of force ($\mathbf{F}$) and the function of speed and position of the object. Therefore, solving this analytically is hard, or even impossible.

To be able to simulate those functions the simulation needs the assumption that the acceleration is constant in a specific time interval, which is only true when the force does not change during this small time. Although this assumption might not hold in reality, if $\Delta t$ is small enough the numerical method approximates the actual function.

One option is to directly use equations (4, 5, 6). This is called the Euler method. For illustration, take the example of an object at rest at position $(0, 0, 0)$, weighing $1kg$, with a delta time of one second ($\Delta t = 1s$) and only the force of gravity of $2n$ acting on the object. Using the previously mentioned functions the following result can be calculated:

$$\mathbf{a}_{y1} = \mathbf{F}_{y0}/m = -2/1 = -2 \tag{7}$$

$$\mathbf{v}_{y1} = \mathbf{v}_{y0} + \mathbf{a}_{y1} * \Delta t = 0 + -2 * 1 = -2 \tag{8}$$

$$\mathbf{s}_{y1} = \mathbf{s}_{y0} + \mathbf{v}_{y1} * \Delta t = 0 + -2 * 1 = -2 \tag{9}$$

This is the next height of the object. To calculate the full state of the object, this should also be done in the other dimensions. After computing this new state of the object, subsequent states can be calculated. This process of using the previous state to calculate the next state can be repeated to run the simulation. Nevertheless, this way of calculating the states is not accurate because the Euler method is only a first order numerical method. To get an accurate result from the Euler method, it needs a extremely low $\Delta t$. This makes it computationally expensive to use. Since this research paper is implementing a real-time physics engine, a better method needs to be found.

Euler method is the simplest member of the Runga-Kutta family. This family of methods contains multiple numerical methods that vary in complexity and accuracy. This is a delicate trade off that needs to be considered. Eulers method is by far the simplest, but also inaccurate. Therefore, the Runga-Kutta of the fourth order is most widely used, as it gives a good trade-off between the complexity of calculating the result and the accuracy of that result. The actual derivation of the Runga-Kutta method is beyond the scope of this paper, but further details can be found online[1].

[1] https://en.wikipedia.org/wiki/Runge-Kutta_methods

# 3 Related Work

This work implements the architecture proposed in “Defeating lag in network-distributed physics simulations"[2] (see Figure 1). This section will only give an overview of the most important points. Further details can be found in the original paper.

## 3.1 Event based system

The architecture uses an event-based system. Every event has a specific time at which it should be executed, and this paper uses a priority queue to sort these events. The edges on the event graph show the scheduling of following events. This could contain complex checking of overlap, or just be a guaranteed scheduling of an event.

The startMove event is the most interesting of the events, since it is the event which happens most frequently. This event moves an object by $\Delta t$. This is done by using the Runga-Kutta method as mentioned in the previous section. Do also note that this event has an extra limitation. Only one startMove event may exist at any one time for a physics object.

## 3.2 Collision checking

For every startMove event the simulation should check for collisions since it wants to change the position of the object if a collision has happened. The original paper proposes a complex system of data structure to optimize the collision checking process. These optimization are required when making a full physics system. However, since we only tested it with a few physics objects, these optimizations are not necessary. Since the performance penalty of this collision check is negligible with one or two physics objects, collision checking can be incorporated into the startMove event.

# 4 Methodology

To evaluate whether lag was reduced in our system, two analytical tests were conducted.

## 4.1 Test 1: same machine

The first test runs on a single machine, simulating the effects of two separate instances of the same custom physics engines running at the same time. This will be the control test, to check whether there is any error in the system if there is no lag. A sphere is dropped from a specified height, which will then bounce due to the physics engine. After the sphere has come to rest, the system will automatically teleport the sphere to a height again. This sphere will also be replicated to the other physics system. Afterwards the height of the spheres can be compared to calculate the error between the two spheres.

## 4.2 Test 2: separate machines

The second test runs on two separate machines on a local network. This way there is a significant increase in the latency, but no complex setup is needed. During this test instead of automatically changing the height of the spheres, the author will use a VR headset to manipulate the sphere. In this way the actual full system is tested. Both machines record the movement of the ball, such that it can be analysed afterwards.

# 5 Implementation

This section describes the choices made for the implementation use in our evaluation. This implementation has been made in Unity version 2022.3.32f1.

## 5.1 Subset implemented

This paper implements the event system. This is the physics system with simple collisions, together with basic networking. The following elements from the original paper were not implemented: the original collision events, the motion witness capsule, and the R*3D tree. This is mainly due to the fact that these elements are there to optimize the physics engine. And it does not change the response of the architecture since they only optimize the performance.

## 5.2 Multiplayer Architecture

An architectural choice that has been made is that one player will function as the server. The server should be able to handle more than one other connected player at the same time. This will not be tested in this work, since that is outside of the scope of this research paper.

## 5.3 Physics system

The physics system runs on a different thread than Unity's main thread because the simulation needs a different frequency than Unity's physics system. The physics system also needs to be able to call events at any time. Both reasons have to do with how Unity's *Update* and *FixedUpdate* are called. *Update* is called once per frame, and *FixedUpdate* is called every physics tick which is defaulted to every 0.02 seconds. Although these settings can be changed it will also cause the Unity physics system to consume more system resources if the $\Delta t$ is reduced.

The event system also needs to be able to execute an event at any time, which is impossible with the *FixedUpdate* calls. If for example the simulation wants to execute two events that are spaced 0.01 seconds apart, then if they are called in *FixedUpdate* that entails that they will still happen on the same physics tick. Therefore, the simulation needs its own thread.

This does cause some difficulty propagating the information back to Unity, since a different thread cannot manipulate Unity transforms. The implementation therefore still only updates the transforms on *Update*, so that once the frame has been drawn, Unity uses the latest information from the physics engine.

## 5.4 Prediction

To ensure that the state is similar on multiple computers, the system implements a basic form of prediction. Once a message arrives that external events have ended, the system simulates the objects for the same amount of time as the network latency between 2 computers. This means that both systems will have almost the same state. This simple form of prediction can be done, since only one moving physics object is used during testing. This means that the full state of the system is sent during testing. And since the full state is sent, no rollback is needed on the other machine. If, for example, the test included two moving physics objects, and the system only sends data about the physics object with external events, then

the receiver of the external events needs to check whether the two moving physics object collided at the moment of the external event.

# 6 Evaluation

## 6.1 Test one: same machine

Figure 2 shows the first part of this test. Here the two spheres were dropped with the same initial conditions. Since both physics engine start at the exact same time, the error is zero.

In Figure 3 the spheres are again placed at a specific height, but only on one physics engine. The other physics engine is updated by using the network code. This way, there is a tiny bit of latency between the physics engines, which results in a tiny difference between the two spheres.

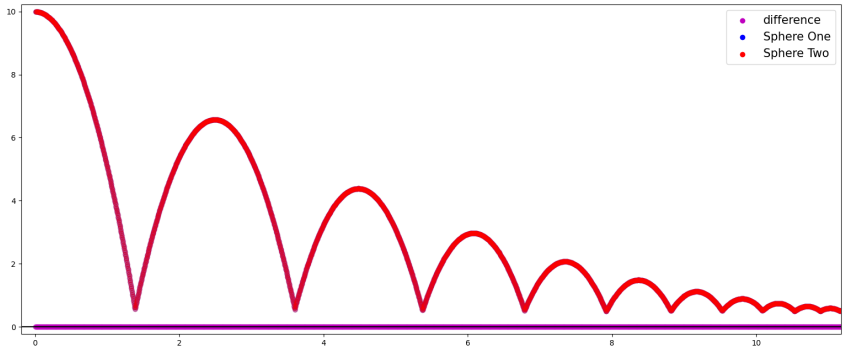


**Figure 2:** The difference and trajectory in height between 2 spheres running on the same machine, starting from the same initial conditions at the same time

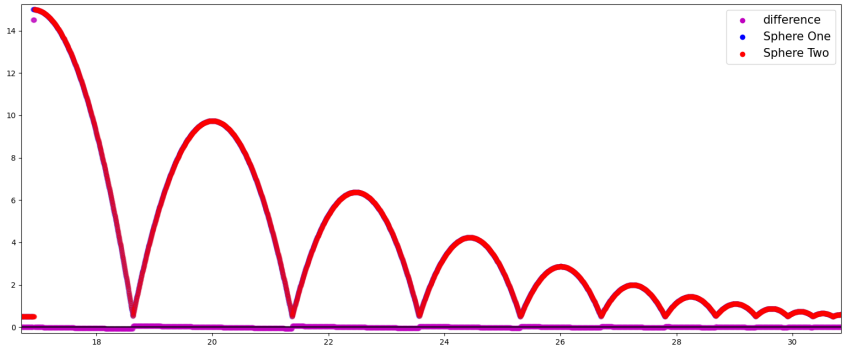


**Figure 3:** The difference and trajectory in height between 2 spheres running on the same machine, starting from the same initial condition but not at the same time due to network lag.

## 6.2 Test 2: separate machines

In the second test we can see the error when moving the ball on two separate machines. Figure 4 shows that if there is some external input, the error is high. But the system can recover once the external input stops. The error afterwards is not completely zero, but this error is not that noticeable once in the application.

Figure 5 shows what happens if there is allot of external input. We can see that lag cannot be reduce in this scenario, since the other system does not know what external input it can expect.

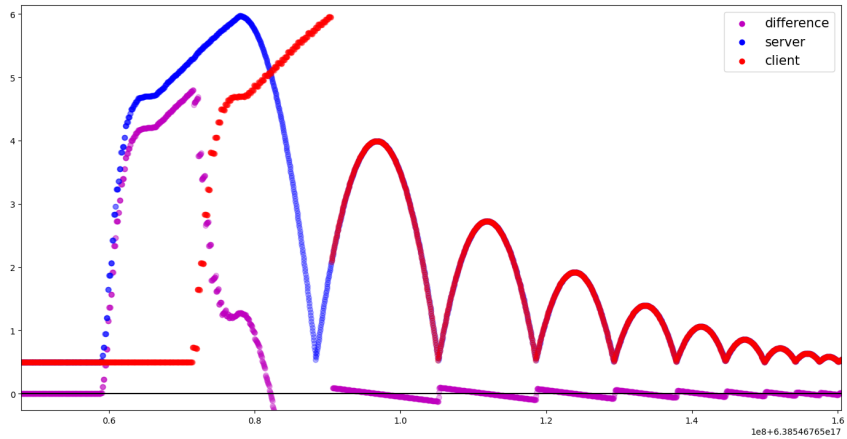


**Figure 4:** The difference and trajectory in height between 2 spheres running on different machines. The first part is caused by the external events, and afterwards the system predicts the updates needed.

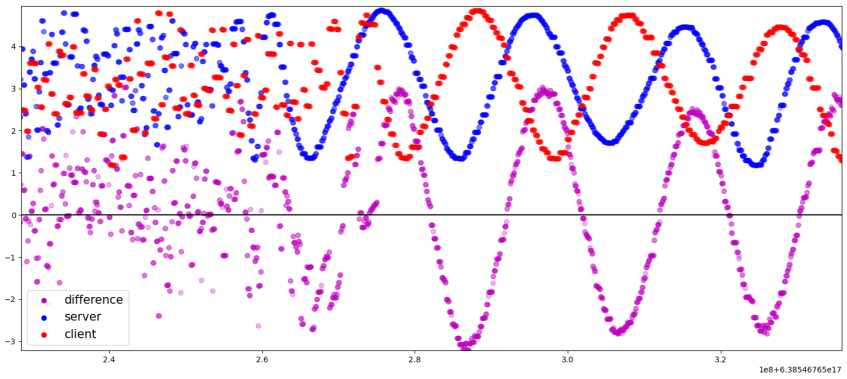


**Figure 5:** The difference and trajectory of two spheres whilst having a lot of external inputs. The first part is caused by randomly changing the position of the ball, and the final part is slowly oscillating the ball.

# 7 Responsible Research

## 7.1 TU Delft

TU Delft has not pushed us to publish positive or negative results. They even encourage us to publish negative results, as that is just as valuable as positive result. Having said that TU Delft will give a grade for this research paper either way. This might make positive results more attractive since they might elicit a higher grade. This is therefore a potential bias. This bias has been mitigated by acknowledging these facts, and still reporting the negative results.

## 7.2 Reproducibility

To make this research reproducible, the code has been published to a public repository[2]. This means that others can freely use this code and are able to use it to reproduce the results.

# 8 Limitations

The program developed for this research paper only implements a subset of the architecture. Hence, only simple simulations can be tested properly.

# 9 Discussion

As the results show, there is a significant difference in state between clients if the system receives external events. There is little lag if there are no external events, and the full state of the simulation can be predicted. The lag during the external events exists, because those external events will always come unexpected to the physics system. This will happen often in an interactive project. Those external events are also hard to handle for the physics system since it will need to 'heal' back to a correct state. This 'healing' process only briefly mentioned but not discussed in the original work[2]. Further research needs to be done on this part of the architecture to find out how to handle this 'healing' process.

Specific attention needs to be made to the following two aspects of the 'healing' process. The definitions of *latency compensation window* (LCW) and the *completed event list*. Both concepts are loosely defined in the original paper and must be well defined to implement a good healing process.

The LCW can be interpreted in two ways. It could be the amount of time the simulation precomputes forward, or it could also incorporate the amount of history the simulation keeps in memory. It is a range of time for which the simulation wants to know the states of all objects in the simulation. It is unclear what the temporal range of the LCW looks like, [0 ms, 100 ms] or [-100 ms, 100 ms]. The original paper does not clarify which of these options it is. It only states that the LCW should depend on "the latency between participants"[2].

If the simulation wants to incorporate the history of the simulation, the paper also implies that there should be a *completed event list*. To implement this feature, it is necessary to know how to store a history of events, and what the size of this history should be. This is

[2]https://github.com/timzonb/bachelor-reducing-lag

needed because the simulation cannot keep the entire event history in memory. It could be based on the LCW, but the paper does not specify any details of how this can be done.

# 10 Conclusions and Future Work

The method proposed by Loren Peitso and Don Brutzman[2] makes latency unnoticeable unless external inputs are present. While this is useful for non-interactive applications, Virtual Reality applications or other games can expect external inputs often and would mostly rely on the 'healing' process to mask lag. The paper of Loren Peitso and Don Brutzman[2] does not fully explain how to do this. Even still, further research should be made to implement those features, since the architecture seems promising.

# 11 Acknowledgements

In this section multiple acknowledgements will be made to sources used in the making of this research paper.

## 11.1 Networking code

The networking code has been used from a tutorial series by Tom Weiland[3].

## 11.2 Peer review

Thanks to Ricardo Marroquim, Amir Zaidi and 4 anonymous TU Delft students for peer reviewing my paper.

# References

[1] Yahn W Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, volume 98033, 2001.

[2] Loren Peitso and Don Brutzman. Defeating lag in network-distributed physics simulations. *Graphical Models*, 111:101075, 2020.

[3] L. Zheng and X. Zhang. Chapter 8 - numerical methods. In Liancun Zheng and Xinxin Zhang, editors, *Modeling and Analysis of Modern Fluid Problems*, Mathematics in Science and Engineering, pages 361–455. Academic Press, 2017.

[3] https://youtube.com/playlist?list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5