DISTRIBUTED COLLISION DETECTION AND RESOLUTION

by

Ching Ling Tom Chen

School of Computer Science

McGill University, Montreal, Quebec

May 2010

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Science

Abstract

Multiplayer, online computer games often distribute game-object state to client machines in order to improve game scalability and responsiveness. Network *latency* and *jitter* are concerns in this context, although the impact is reduced by the use of predictive techniques such as *dead reckoning*. These techniques, however, introduce consistency concerns for important and hard to predict behaviours, such as object collisions. In these cases a centralized authority or client/server architecture is typically used to ensure strong consistency, limiting game scalability.

In this work we propose a *motion-lock* protocol for distributed game collision detection and resolution. The motion-lock protocol improves performance of motion prediction by giving stations time to communicate and agree on the detected collisions. This reduces the divergence of object states and post-collision trajectories. Offline and online simulation results show the motion-lock protocol is able to maintain strong consistency in collision count and reduces post-collision deviation with a small sacrifice of 3-4% in responsiveness of player controls. Qualitatively, the visual result of the collision response is greatly improved. With the motion-lock protocol, multiplayer online games can offload basic collision detection and resolution to game clients, increasing scalability without overly sacrificing consistency.

Résumé

Multijoueurs, des jeux informatiques en ligne distribuent souvent tat du jeu-objet aux ordinateurs clients afin d'amliorer l'extensibilit de jeu et ractivit. Temps de *latence* et la *gigue* sont des proccupations dans ce contexte, bien que l'impact est rduit par l'utilisation de techniques de prdiction, comme *dead reckoning*. Ces techniques, toutefois, d'introduire des proccupations importantes pour la cohrence et difficile prdire les comportements, tels que les collisions d'objets. Dans ces cas, l'architecture d'une autorit centralise ou client/serveur est gnralement utilis pour assurer la cohrence forte, limiter l'volutivit jeu.

Dans cette texte, nous proposons un protocole de *motion-lock* pour la dtection de collision jeu distribu et de la rsolution. Le motion-lock protocole amliore les performances de prdiction de mouvement en donnant stations temps de communiquer et de s'entendre sur les collisions dtectes. Cela rduit la divergence des tats de lobjet et trajectoires post-collision. Les rsultats de simulation en mode den ligne et hors ligne montrent le protocole motion-lock est en mesure de maintenir la cohrence forte du nombre de collisions et rduit l'cart aprs la collision avec un petit sacrifice de 3-4% de la ractivit des commandes du lecteur. Qualitativement, le rsultat visuel de la raction de collision est grandement amliore. Avec le protocole de motion-lock, les jeux multijoueurs en ligne peut dcharger la dtection de collision et la rsolution de base aux clients de jeu, ce qui augmente l'volutivit sans trop sacrifier la cohrence.

Acknowledgments

I would like to thank my supervisor Professor Clark Verbrugge. He allows me to freely explore this new research area, while, at the same time, guiding me to the right direction. His valuable advice and patience have made this thesis possible.

I would express my gratitude to my family. Encouragements from my parents have kept me going through difficult times during my graduate studies.

Finally, I would like to give special thank to Pauline Lau for believing in me and my decisions. Without her support and understanding, this thesis would not be completed.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Multiplayer online games (MOG) have become popular in recent years. Advances in networking technology allow game developers to send more data with faster transmission and thus support larger and more complex virtual worlds. Currently, most multiplayer online games out in the consumer market use the centralized client/server architecture as a basis for the game implementation [12, 11]. This architecture is relatively easy to implement, and the centralized server helps the companies deal with issues such as billing and security. A single game authority further simplifies many game implementation aspects, such as in modeling game physics. For these interactions a centralized authority is useful in order to ensure detection and resolution of important game events, such as object collisions, are the same for all clients.

As the game population increases, however, basic client/server models suffer from the server acting as a bottleneck to game state processing and communication. The recent success of Massive Multiplayer Online Role Playing Games (MMORPG), which emphasize large and growing player populations, has motivated the game companies to look into more scalable architectures. Peer-to-peer and more complex hybrid designs show promise [14, 7, 1]. Unfortunately, as game sizes grow and as information is distributed, state consistency becomes more difficult to maintain, reducing either performance or apparent accuracy and thus interfering with immersive game-play.

Techniques exist to improve scalability while maintaining real-time performance, and in the academic and the military community research into the closely related

field of distributed virtual simulation has been around since the 1980's. An important solution to scalability in such contexts is the use of optimistic techniques that locally cache or *replicate* remote-object data. Stations can then interact optimistically using local versions, avoiding the need to always consult with the holder of remote data. When data changes, synchronization of replicas is performed by sending messages between replicas and the original *master*. Network latency and jitter can of course delay updates, causing gaps or delays in representing or resolving state. To overcome network problems, *dead-reckoning* algorithms are used to predict object motions and interpolate any missing data. This reduces round-trip communication delays, at a cost of reduced accuracy in representation.

Although these designs improve scalability, they have a less positive impact on the representation of important interactions such as collision detection or resolution. Dead-reckoning is least successful in the presence of unpredictable, dynamic interactions such as game-object collisions. Errors in dead-reckoned motion can affect the time, location, and even detection of collisions, easily resulting in large visual errors as the game state deviates and is eventually synchronized. Figures 1.1 and 1.2 show examples where errors introduced by dead-reckoning cause a collision to be missed or detected erroneously (respectively). The potential for this behaviour is an important aesthetic concern in client/server architectures, and has an impact on overall game consistency in peer-to-peer contexts.

Here we investigate a new protocol for improving collision consistency between pairs of distributed objects. Our *motion-lock* protocol works by observing object behaviour and preventing unpredictable local object movements in the presence of potential collisions. By matching limitations on object activity to network delay, the prediction of future collisions can be greatly improved, optimally resulting in in perfect collision synchrony. This simple design has few drawbacks, adding only minimal additional network cost and having little to no discernible user impact.

We evaluate our design experimentally, measuring and comparing behaviour to an industry-standard dead-reckoning design, as well as in relation to basic optimistic and pessimistic designs for improving collision consistency. Our design shows significant benefit to game consistency, greatly reducing the inconsistency times over more
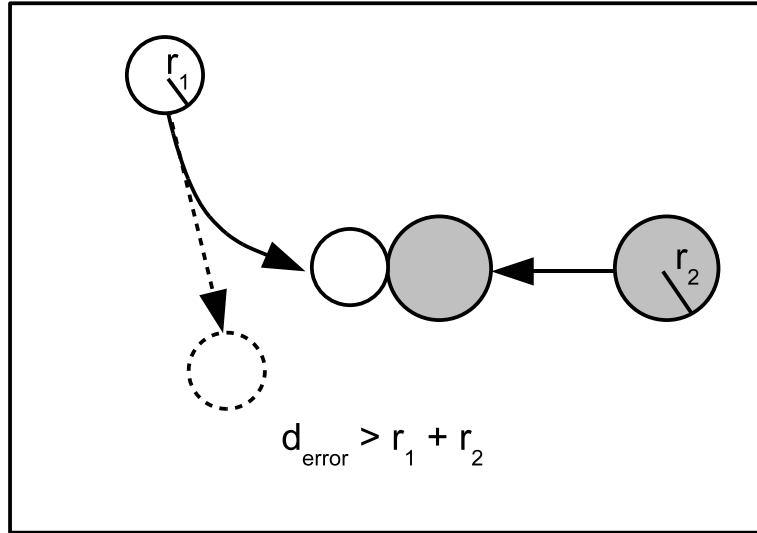
Figure 1.1: A missed collision when error in the location of one object is greater than the sum of the radius of the objects.
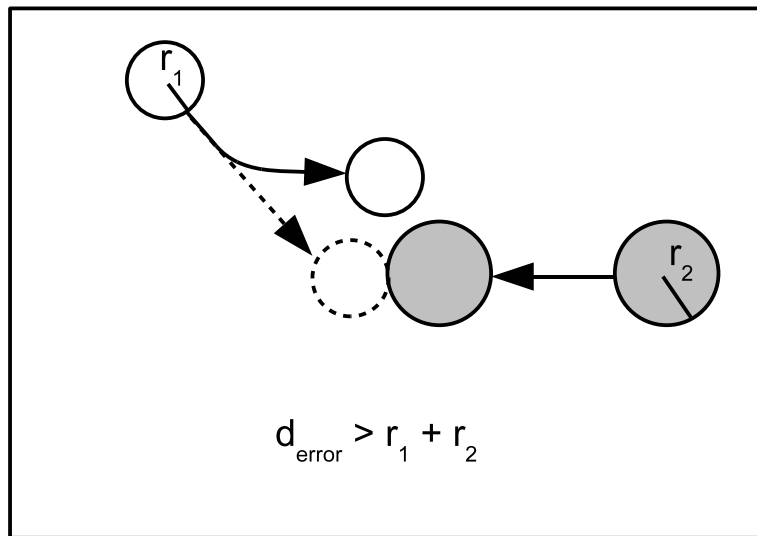
Figure 1.2: A false collision when error in the location of one object is greater than the sum of the radius of the objects.

straightforward approaches on average, and noticeably improving visual appearance.

1.1 Contributions

Our work makes the following specific contributions:

- We develop a new protocol for improving collision consistency in a distributed, peer-to-peer environment. Our approach constitutes a form of *local lag* [19] specialized to collision handling, and has low network and user impact.

- To help evaluation, we develop two implementations of our design. These consist of an offline simulation capable of arbitrarily varying test parameters, and for more complex and realistic testing an implementation within a full-featured, industry-grade game network middleware.

- As part of validating our design we perform detailed experimentation, showing data comparing the behaviour of our approach under both offline, simulated and online measured implementations. We consider both qualitative and quantitative characteristics relevant to our problem. Our motion-lock protocol has demonstrably better behaviour, reducing inconsistency time and improving visual appearance, while also demonstrating no significant network overhead.

In the next chapter we give essential background and related work for understanding our design and approach to distributed collision detection. Chapter 3 examines the challenges in distributed collision detection in detail, and defines important definitions that will be used throughout this thesis. Chapter 4 describes our main protocol design, and Chapter 5 gives experimental data evaluating our design. We conclude and describe future work in Chapter 6.

Chapter 2
Background and Related Work

Traditionally, a multiplayer offline game consists of a single station with a single instance of game state. Players enter commands through different controllers to manipulate the game state on the single station. At its core, a multiplayer online game (MOG) is similar to its offline version where there is a shared game state. However, players are distributed geographically and manipulate the game state on their own local machines through the network.

In an MOG there are two main types of architectures used to maintain the shared game state: centralized and distributed. The centralized architecture contains a single instance of game state that resides on a server station. Players send commands from their local client machine to the server to change the shared game state. The server then processes the commands, calculates a new game state and sends the new state back to the clients. Since there is only a single game state, the centralized architecture guarantees strong consistency of the game state. However, the server has finite processing power and bandwidth to handle a given number of clients, and so this architecture has difficulty scaling up to large game sizes. Furthermore, if the server crashes, the game state is lost and the game is over.

In the distributed architecture, such as peer-to-peer and server cluster architectures, there are multiple instances of the game state replicated on different stations. A player's local station can either retain its own instance of game state, or connect to one of the stations with an existing replication. If one station crashes, there are still

5

other stations maintaining the shared state that allow the game to continue. Player commands can (optimistically) change one of the instances, and thus, the processing of the game states is distributed, avoiding the single server bottleneck. A disadvantage of this design is that in order to keep the instances synchronized, the stations need to send update messages to each other. If the network conditions between stations becomes congested, however, updates can be delayed or lost, causing the states to diverge and thus become inconsistent.

In this chapter, we first present some background on the distributed architectures of games and how game state is replicated and updated. Next, related work on consistency and collision handling in multiplayer online games is presented.

2.1   Consistency in Distributed Architectures

In modern multiplayer online games, thousands of players can connect to the game and interact with each other. With a client/server architecture, the centralized server contains the single game state and maintains the consistency of the game state. In a distributed architecture, the game state is replicated on different stations so that the processing of the game state is distributed. However, replication introduces inconsistency. In this section, we discuss the basics of game state replication and the cause of inconsistency. Then we show some pioneering works in distributed virtual simulation and how they handle the inconsistency.

In a game, the entire game state can be broken down to individual states for each object. For example, the state for an item that a player may or may not possess, would include whether it is acquired by a character or not. In a distributed environment, each object and its state may be created by one of the game instances on one station. The object is then required to be replicated on other stations so that all instances which may need to represent or otherwise deal with the existence of that object have the same state.

In our research we focus on the physically based motion state of a player-controlled

object. The motion state of a controlled object is its position, velocity, and acceleration. We assume the state can be changed by player inputs, but also follows an underlying motion model. Objects thus undergo change not only from player inputs but may also experience continuous movement or state change due to the motion model.

In order for other players to see each other's controlled objects, the controlled objects are replicated on all players' local stations. For our discussion, we term the copy of the object that is controlled by the player the **master**, and the replicated copies that are located on other players' remote station the **replicas**. On each station, players can control their own master object and interact with the replicas.

SIMNET [5] and DIS [2] were the pioneers in distributed virtual simulation developed by the United State Defence Advanced Research Projects Agency. SIMNET was used to train military personnel in warfare tactics and operating various vehicles. The protocol used in SIMNET was later on standardized to the DIS protocol. In DIS, the controlled objects operated by the soldiers are replicated on all stations. The motion states of the master objects are sent through the network to update the state of the replicas on the other stations. Unfortunately, network latency is always present in a distributed simulation. Master updates will be delayed and the replicas will be perceived to be moving in the past. The states of the replicas are not synchronized with the master and any interactions with the delayed replicas cause inconsistency.

To compensate for network latency and possible packet loss, DIS uses a *dead reckoning* algorithm to synchronize the states of the masters and their replicas. In general, dead reckoning algorithms use the past state to extrapolate a future state based on the underlying motion model. For example, if an object is moving according to Newton's law of motion, its future position can be predicted using the equation $P_{t+\Delta t} = P_t + V_t \Delta t + A_t \Delta t^2 / 2$. In DIS, the dead reckoning algorithm uses the received but outdated master state to extrapolate a current state for any corresponding replicas. The position of the replica is then approximately in sync with the master's position; this reduces the inconsistency when the objects interact with each other.

The DIS dead reckoning protocol also reduces the bandwidth by controlling the update frequency. On the master station, a prediction of replica state is made using

7

the same dead reckoning algorithm as each replica may use. The master station then calculates the difference between the actual state and the predicted state of the master. Only when the difference exceeds some preset threshold will the master station send update to the replicas. In other word, replicas are updated only when the master believes the replicas have a large deviation. In practice this can be combined with the *Position History-Based Dead Reckoning* protocol [28], which improves upon the DIS protocol by further reducing bandwidth usage. Instead of sending position, velocity, and accelerations to the remote stations, only the position of the master is sent to reduce the packet size. Upon receiving a position, the remote station stores the position in a history. As the frame time is reached to display the objects, the remote station uses the positions in the history and an adaptive tracking algorithm to estimate the current velocity and acceleration. With the received position and the estimated velocity and acceleration the current position can be extrapolated.

Although dead reckoning algorithm can help synchronize the state of the replicas with its master, for controlled objects players can enter inputs at anytime to change the master's motion. The inputs are discrete events that cause the master to temporary disobey the motion model and become unpredictable by the dead reckoning algorithm. Similarly, collisions between objects are discrete collision events that cause the master to break away from the motion model. Both player inputs and collisions cause the dead reckoning algorithm to unable to predict the master's state, and so the replicas deviate and create inconsistency.

State consistency in such environments does not grow unbounded. Even if a replica has deviated, its state will be corrected after applying the next state update from the master. This correction, however, can be large, visually apparent and thus confusing to players. The dead reckoning protocols thus also use smoothing algorithms to gradually converge any deviated state to the correct state to produce better visual results. Unfortunately, a smooth transition of the replica's trajectory that is still not in sync with the master's actual motion, may not correspond to the actual object motion model, and may further diverge in state if any other inputs or collisions occur during the transition period.

2.2 Related Work

As discussed above, motion prediction is essential for multiplayer online games to synchronize the motion state of a master and its replicas. The DIS dead reckoning protocol [2] and the Position History-Base Dead Reckoning protocol [28] provide the basis in motion prediction to improve consistency for distributed virtual simulations and multiplayer online games. In this section, we discuss some of the relevant research that provides more consistency to distributed architectures. This research can be divided into consistency in object state, and a higher level consistency in global game state.

2.2.1 Object State Consistency and Motion Prediction

To improve consistency, many researchers have looked into improving the dead reckoning protocol by using alternative measurements or adaptively changing the attributes of the protocol. Cai et al. adaptively changes the error threshold of the dead reckoning algorithm depending on the distance between the object. When two objects enter each other's area of interest such that the distance between them is small, the error threshold is reduced so that the update frequency is increased to reduce prediction error [4]. Similarly, Kenny et al. calculates the deviation error of the replica and sends back the error to the master, so that the master can change the error threshold accordingly. This creates a close-loop control system to adjust the update frequency [15]. The Pre-reckoning algorithm overrides the error threshold and sends updates immediately to the replica if the motion of the master shows the following three behaviours: resting master starts to move, moving master comes to a full stop, and master makes a sharp turn [10]. Robert et al. introduce the time-space threshold to determine the update frequency. The deviation between the master's absolute state and predicted state is summed over time between two successive updates. The time-space threshold performs better than the original error threshold when objects move with smooth and low curvature motions [27].

Similar to dead reckoning algorithms, a Kalman filter estimator can be used to

estimate object motion states. The Kalman filter predicts a state based on the underlying model, and then corrects the prediction based on the measured data. For details on Kalman filter, Welch et al. [32] provides a good introduction. In their research, Tumanov et al. [31] use a Kalman filter to predict the future state of the master on the sender side. The future time of the predicted state is relative to the estimated network latency. The predicted master state is then sent to update the replica such that it will arrive on time without the need of extrapolation at the receiver side.

Chan et al. [6] suggest that most distributed virtual simulations use the 2D mouse to navigate through the environment. They propose that by predicting the 2D movement of the mouse and then mapping the movement onto the 3D virtual world, the prediction of the object's motion can be simplified. They studied the mouse movement and use a Kalman filter estimator to predict the movement. However, this work is limited to games that uses a mouse for control.

Research into motion prediction can help improve the accuracy of state predictions, which, in turn, can reduce the inconsistency caused by collision events. Still, motions for user controlled objects can be hard to predict, and with high latency and jitter false and missed collision can still occur. Therefore, improvements in motion predictions are not enough to maintain consistency in object states when there are collisions between objects.

For collision detection, in his research, Ohlenburg adaptively increases the rate of updates from master to replicas when objects are close to each other and may potentially collide [21]. The results show great improvement in object collisions, but also show that by increasing the update rate, the bandwidth usage increased dramatically. This work is closely related to our research; however, the adaptive approach does not deal with missed or false collisions. Stations will not be informed if a collision is missed. This creates inconsistency in number of collisions detected, as we will discuss in detail in the next chapter.

For more predictable non-player controlled objects, the Deterministic Object Position Estimator uses an object's past trajectories to predict the motion of the object and the collision point and time [17]. The estimator shows accurate results for objects that follows predictable trajectories, but does not apply to player controlled objects

10

with unpredictable movements.

Another approach to improve state consistency is to manipulate the time when events are actually performed. In the *local lag* algorithm [19], player commands that are to be applied to the masters are delayed and scheduled to be executed at a future time. The commands are then sent to the replicas, hopefully arriving before the scheduled execution time. When the scheduled time has reached, the master and its replicas process the commands simultaneously, synchronizing the states and maintaining consistency. Here responsiveness to player commands is traded-off for consistency. The amount of delay (lag) should not be so long as to be perceivable by players, but long enough to allow state synchrony. Our motion-lock protocol is similar to the local lag algorithm such that player's loses control to gain collision consistency; however, the motion-lock protocol only applies the locking when collisions are predicted, while local lag delays all player commands.

2.2.2   High Level Consistency Architectures

In this section, research in maintaining and synchronizing global states is discussed.

Due to variable network delays and differences in frame rates among the connected stations, the game state on each station may not be synchronized. In the research of Diot et al. [9], the bucket synchronization algorithm discretized time into buckets of fixed intervals. Commands are delayed, similar to the local lag but with a fixed 100ms lag, and are put into a bucket to be processed. At the scheduled time, commands that belong to the same bucket are processed at the same time. Missing states due to latency are extrapolated from the old buckets using dead reckoning algorithms. The bucket synchronization algorithm helps stations running at different frame rates to be synchronized and maintain consistency.

However, when critical events such as collisions are delayed and cannot be predicted by dead reckoning algorithms, replicated game states can diverge. The time-warp algorithm [19] [20] can be used to repair the states. Received events and updates are kept in buckets. When a delayed event has finally arrived, with the old bucket, the time-warp algorithm recalculates the current states with the events from the time

11

of the delayed event up to the time of the most current event. The time-warp algorithm not only maintains consistency but also state correctness. However, this algorithm aims at different goals then the motion-lock protocol. Our goal is not to have state correctness but to synchronize collision events among stations to gain state and collision consistency.

Similar to the time-warp algorithm, the *trailing-state synchronization* algorithm [8] rolls back and recalculates the game state when there are inconsistencies. The algorithm uses the mirror server architecture that replicates the game states on different servers with different locality. Comparing to the time-warp algorithm, instead of keeping a series of snapshots of the game state in old buckets, multiple copies of the game with different delays in simulation time are running simultaneously on the mirrored servers. Whenever a rollback is required, the trailing game state that is currently running with a delay such that it has data just before the inconsistency is used to recalculate the current game state.

Another high level approach to maintain consistency is the use of hybrid architectures. Hybrid architectures combine the centralized and distributed architecture that provides good scalability in peer-to-peer systems, while still providing some form of centralized authority over the game states. The Zoned Federation architecture [14] is a hybrid architecture that divides the game space into zones. Each zone is maintained by a federation of nodes containing the zone owner and the zone members. Each zone member can hold some data for the zone. In order to modify the zone data, members need to request an update from the zone owner. The zone owner serializes the requests and ensures consistency of the zone. In a similar approach, SimMud [16] divides the game based on a player's locality in the game world. Players in the same region broadcast to each other to update their state. To ensure consistency, shared objects are assigned to a coordinator. The coordinator updates and resolves conflicts of the object it governs.

While most architectures divide the game world spatially, in Ghost [29], the game space is divided into different latency groups. Nodes that are experiencing the same network latency are allowed to interact and exchange game events. On the other hand, nodes from different latency groups are not allowed to interact. By grouping nodes

into latency groups, nodes with high latency will not hinder the performance and consistency of nodes with low latency. Ghost provides good consistency and avoids issues of unfairness that can result from players with better or worse connectivity, but limits the freedom of players in choosing which other players to interact with.

2.3   NetZ Middle-Ware

To prove that the protocol developed in our research can be used in real industrial grade multiplayer online games, we implemented and tested our protocol within the *NetZ* middle-ware. NetZ is a distributed state management framework written in C++. It is developed by Quazal and has been used successfully in many popular games in the gaming industry [30].

The NetZ middle-ware is based on a distributed architecture. At its core, a unique *Data Definition Language* and *Duplication Space* model defines the objects and replicates them among the stations. The Data Description Language uses a C++-like syntax to define the type and name of the data to be sent over the network. The Data Definition Language compiler compiles the data and generates optimized C++ code for marshalling/unmarshalling and sending/receiving the data. The Duplication Space technology uses a publish-subscribe model to automate the object replication process. NetZ implements a number of features to help synchronize game objects and maintain consistency; this includes PHBDR, Local-Lag, Bucket-Synchronization, and more [26, 25, 24]. Game developers can easily turn on and off any of these algorithms to meet the needs of their game.

For collisions, NetZ provides a *Local-Correction* algorithm, allowing replicas to resolve collisions independently on the remote station based on local data. When the remote station has detected a collision between a replica and some object, instead of updating the replica with the predicted states calculated by the dead reckoning algorithm, the station resolves the collision and calculates a post-collision trajectory for the replica. This helps remove object penetrations when master updates are delayed. Chapter 3 provides a more detailed discussion of this issue.

Chapter 3
Challenges

Network latency and packet loss are two main challenges encountered in multiplayer online games and distributed virtual simulations. Network latency and packet loss reduce the ability of such applications to maintain consistent and correct application states among the stations. For player controlled objects in multiplayer online games, network latency delays state updates from the masters to their replicas, and packet loss prevents replicas from receiving all state updates from the masters. These problems affect the synchronization of the states of the replicated objects and ultimately cause problems for collision detection and resolution.

In Chapter 2, we see that many algorithms and protocols have been developed to predict the motion states of controlled objects. Dead reckoning algorithms are widely used in multiplayer online games because they are effective and easy to implement. However, as network latency and packet loss increases, dead reckoning algorithms can produce large extrapolation errors, increasing consistency concerns.

In this chapter, we take a closer look on how network latency and packet loss affects motion prediction and how they lead to extrapolation errors. The chapter then examines how the extrapolation errors affect collision detection and resolution in distributed systems. Finally, the chapter formalizes consistency and correctness for motion prediction, distributed collision detection and distributed collision resolution.

3.1 Challenges in Motion Prediction

Because of network latency and packet loss, replicas may not know the absolute state of their master. The purpose of motion prediction in multi-player online games is then to predict states for the replica using past states to compensate for the effects of network latency and packet loss. However, for controlled objects, player inputs can be entered to change the state of the masters, and thus, the masters' motion cannot be predicted accurately. Furthermore, as network latency and packet loss increase, the prediction error increases. To understand how player inputs affects motion prediction, the basics of motion prediction is first presented. The problems caused by the combination of unreliable network and player inputs are then examined in detail.

3.1.1 Motion Prediction

It is possible for a replica to accurately predict the motion of its master if the master's trajectory can be described by a certain motion model, and the model and starting point of the trajectory are known to the replica. For example, many futuristic first-person-shooters have a "rail-gun" as one of the weapon choice. Slugs fired out from a rail-gun follow a well-defined, coiled trajectory. If the motion model of the coiled trajectory and time of fire are known ahead of time, the location of the slug can be accurately predicted at any time, irrespective of network latency and packet loss.

To understand motion prediction for controlled objects, we first define the state of the object at time t as a vector:

$$X_t = \begin{bmatrix} P_t \\ V_t \\ A_t \end{bmatrix} \tag{3.1}$$

where P_t is the position vector, V_t is the velocity vector, and A_t is the acceleration vector. P_t, V_t, and A_t contain the x, y, and/or z components, depending on the dimensionality of the game.

Next, the motion model that describes the kinematics of the object is the equations of motion:

$$\hat{P}_{t+\Delta t} = P_t + V_t \Delta t + A_t \Delta t^2/2 \tag{3.2}$$

$$\hat{V}_{t+\Delta t} = V_t + A_t \Delta t \tag{3.3}$$

where $\hat{P}_{t+\Delta t}$ and $\hat{V}_{t+\Delta t}$ are predicted position and velocity at time $t + \Delta t$. The equations of motion are functions of time that assume constant acceleration. Dead reckoning algorithms are based on this model. The motion model can be represent by matrix multiplication:

$$\hat{X}_{t+\Delta t} = FX_t \tag{3.4}$$

where F is a 3 by 3 matrix that represents the equations of motions:

$$F = \begin{bmatrix} 1 & \Delta t & \Delta t^2/2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

To extrapolate a state for a replica, dead reckoning algorithms use the most recently received state as the initial state, X_i. To extrapolate a state at the current frame time, the extrapolation interval Δt_{ex} is calculated by subtracting the current frame time from the time of the initial state. $\hat{X}_{t+\Delta t_{ex}}$ can then be calculated using equation 3.4, where $\Delta t = \Delta t_{ex}$ in matrix F. Figure 3.1 shows the extrapolation of state $\hat{X}_{t+\Delta t_{ex}}$ for the replica using the most current received state X_i. If the master moves at constant acceleration during Δt_{ex}, the motion model can accurately extrapolate a state at the current frame time for the replica.

3.1.2 Extrapolation Errors

For a controlled object, a player's inputs change the acceleration of the master, and as stated above, a replica's extrapolated state is accurate if the acceleration is constant. Changes in acceleration and direction can be seen as an impulsive state change that do not have any past state to relate to, and thus cannot be predicted. The predicted states are therefore sometimes erroneous.

16
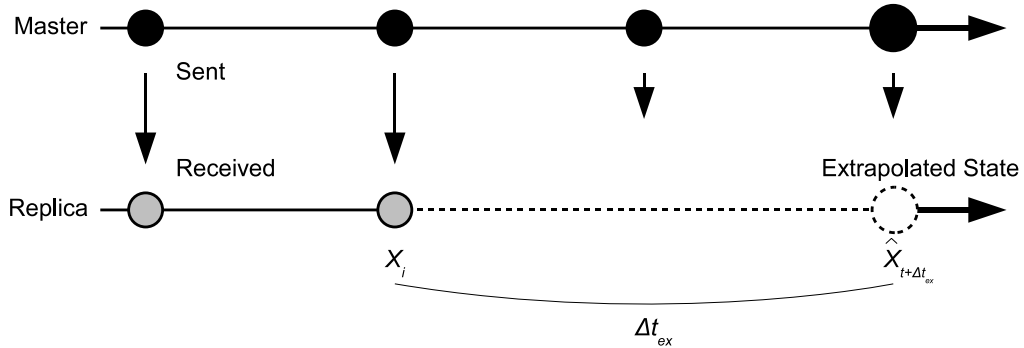
Figure 3.1: Accurate Extrapolation

If the extrapolation interval Δt_{ex} is short, the error is small and may not be apparent when rendered. However, as network latency and packet loss rate increases, the dead reckoning algorithm requires the use of older received states to extrapolate and thus Δt_{ex} increases. If the replica is already traveling at an inaccurate trajectory, as Δt_{ex} increases, the replica will continue to drift away from the master's trajectory. Furthermore, players have more time to enter more input during the interval to changing the state of the master. The replica then becomes more uncertain of the state of the master. Figure 3.2 shows the effect of player inputs changing the master's trajectory during Δt_{ex}. The replica is unaware of the inputs, and so the extrapolated state is inaccurate.

Extrapolation errors lead to different states between the master and the replicas at a given time. If the master and the replica are at different states, interactions with the replicated object may lead to different results on different stations. This causes inconsistency among the stations.

One way of measuring the extrapolation error is by measuring the distance between position of the master and the replica. We denote this as deviation. Assume the controlled objects are represent by circles with radius r. In terms of collisions between objects, if the deviation is less than $2r$, part of the replica may still collide with an approaching object and resulting both the master and the replica encountered the collision. However, if the deviation is greater than $2r$, the replica may dodge the approaching object and miss the collision. Therefore, deviation of the replica that is

17
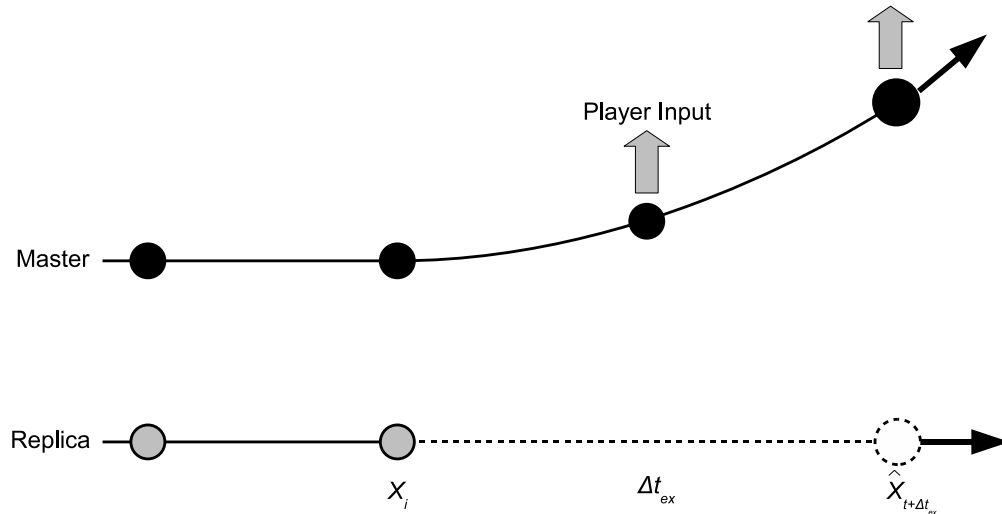
Figure 3.2: Unreliable Extrapolation

greater than $2r$ is considered significant for collision detection. We will discuss this issue in detail in the next section.

3.2 Challenges in Distributed Collision Detection and Resolution

Collision detection and resolution algorithms require using the states of the colliding objects to correctly detect the collisions and calculate the post-collision trajectories. However, due to extrapolation errors, replicas may not have the same states as the masters. Therefore, in distributed architectures, if the stations are to process the collisions independently, collision detection and resolution algorithms may return different results among the stations: some stations may detect the collisions while some stations may not. For collision resolution, the post-collision trajectories may also be different, at least until the replicas receive a master state update to correct the trajectory. During this interval the stations will be inconsistent and produce confusing results.

In the next two sections, we will describe the inconsistency in distributed collision

detection and resolution in detail. To aid the discussion, we assume there are two stations A and B. A contains master M_1 and replica R_2. B contains master M_2 and replica R_1. All objects are represented as circles. Assume M_1 and R_1 have radius of r_1, and M_2 and R_2 have radius of r_2. Figure 3.3 shows the objects on the stations. Masters are represented with black circles, and replicas are represented with grey circles.
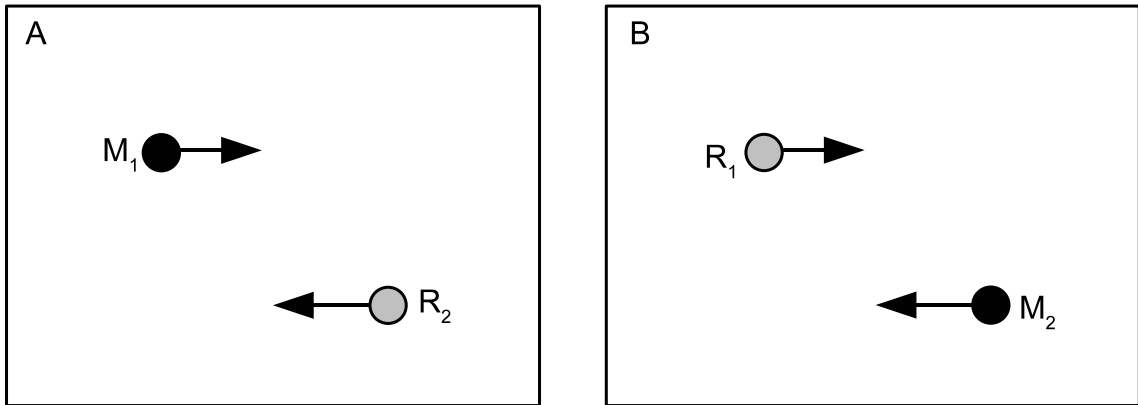
Figure 3.3: Objects on station A and B

3.2.1   Inconsistency in Distributed Collision Detection

As stated above, a master and its replicas may have different states at a given time due to player inputs and extrapolation errors. When the deviation error is large, replicas may unintentionally collide or miss a collision, and the collision detection algorithm returns different results among the stations. The inconsistency can be categorized into two types: false collisions and missed collisions.

False Collisions

A false collision is an unintended collision between a replica and an object on one station, while the same collision is not detected between the master and the object on the station of the master. The problem occurs when there are player inputs that

change the master's trajectory, and the replica extrapolates a position with a large deviation error that causes a collision with other objects.

Using the above setup, assume at time t_0, M_1 and M_2 are moving toward each other without the players entering any inputs. R_1 and R_2 are then moving accurately without extrapolation errors. As the objects are about to collide with each other, player 1 decides to avoid the collision by changing M_1's trajectory at time t_{10}. On A, M_1 successfully avoided the collision with R_2. At t_{20}, M_1 and R_2 are d distance away from each other, centre to centre. On B, however, because of network latency and packet loss, R_1 did not receive the update at t_{10}. At t_{20}, R_1 extrapolates a state where deviation $d_{err} \geq d - (r_1 + r_2)$. The extrapolation error causes R_1 to collide with M_2 and results in a false collision on B. A is correct for not detecting any collision while B is incorrect and inconsistent for detecting a collision.

Figure 3.4 shows the above scenario. The dashed arrow and circle are the extrapolated trajectory and state. As shown, the extrapolated state of R_1 collided with M_2 on B, while M_1 turned away from R_2 on A.
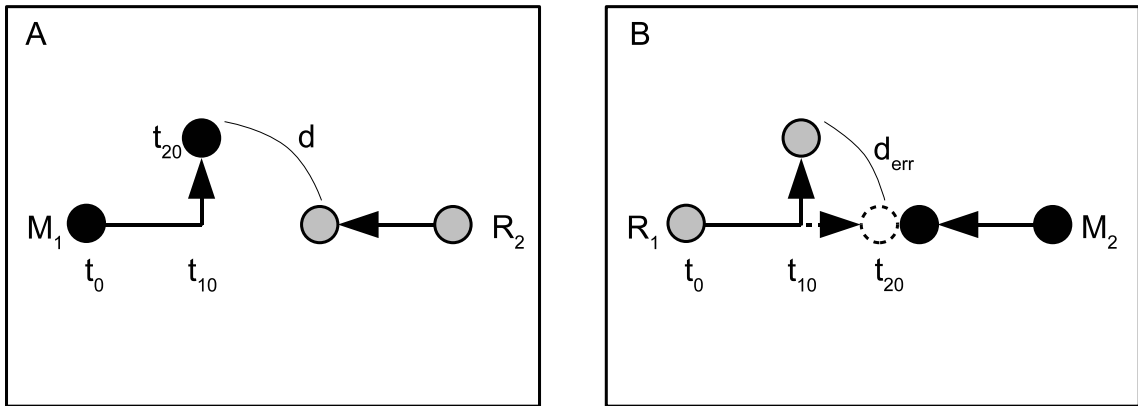
Figure 3.4: False collision causes inconsistency between A and B.

Missed Collisions

A missed collision is an intended collision between a master and an object on one station, while the same collision is not detected between the replica and the object on another station. The problem occurs when the master changes its trajectory to collide

with other objects while the replica did not receive the update and extrapolates a state that misses the collision.

Using the above setup, assume at t_0, M_1 and M_2 are moving towards each other but will not collide, and the players are not entering any inputs. R_1 and R_2 are then moving accurately without extrapolation errors. At t_{10}, player 1 suddenly changes M_1's trajectory to collide with R_2. On A, M_1 successfully collided with R_2 at t_{20}. On B, however, because of network latency and packet loss, R_1 did not receive the update at t_{10} and continues to move on the current trajectory. At t_{20}, R_1 extrapolates a state where the deviation error $d_{err} > r_1 + r_2$. The extrapolation error causes R_1 to miss the collision with M_2. A is correct for detecting a collision while B is incorrect and inconsistent for missing the collision.

Figure 3.5 shows that on A, M_1 changes its direction to collide with R_2, while on B, R_1 continues to move at the same direction and extrapolates a state that completely missed the collision. The dashed arrow and circle are the extrapolated trajectory and state.
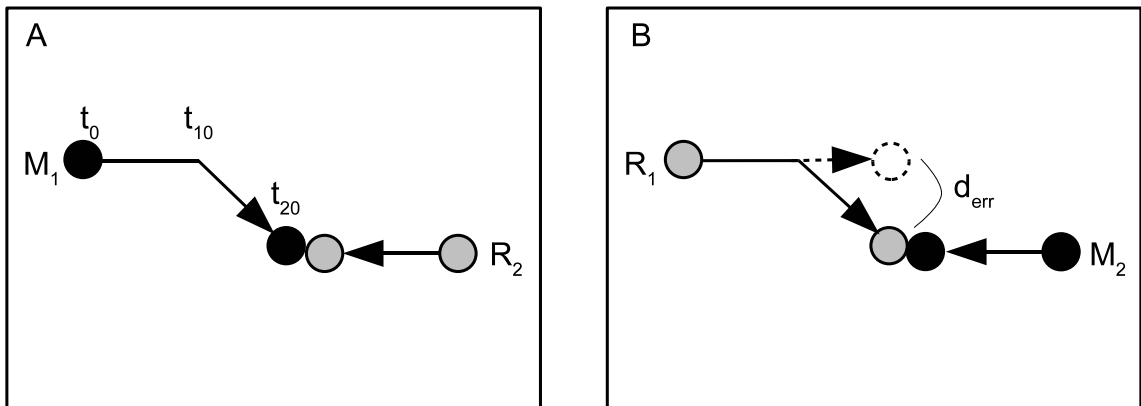
Figure 3.5: Missed collision causes inconsistency between A and B.

3.2.2 Inconsistency in Distributed Collision Resolution

After a collision has been detected, the states of the colliding objects are taken to calculate the post-collision trajectories. However, since a master and its replicas

may have different states due to player inputs and extrapolation errors, at the time of collision, if the deviation between the master and a replica is large the replica may miss the collision, as described in the above section. On the other hand, if the deviation is small, both the master and the replica will collide, but the states taken to calculate the trajectories at each station may be different. Since the states are different, the post-collision trajectories will also be different.

Using the above setup, assume the player 1 frequently enters inputs to change M_1's acceleration and direction. M_1 is then moving unpredictably and R_1 is uncertain about M_1's state. When M_1 collides with R_2, the deviation error for R_1 is $d_{err} \leq r_1 + r_2$. R_1 will collide with M_2, but the states of M_1 and R_1 are not equal. The post-collision trajectories between the masters and their replicas are then different.

Figure 3.6 shows the collision between the objects. The dashed arrow and circle are the extrapolated trajectory and state, and gray arrows are the post-collision trajectories after the collision. As shown, the predicted state for R_1 is different from M_1's state.
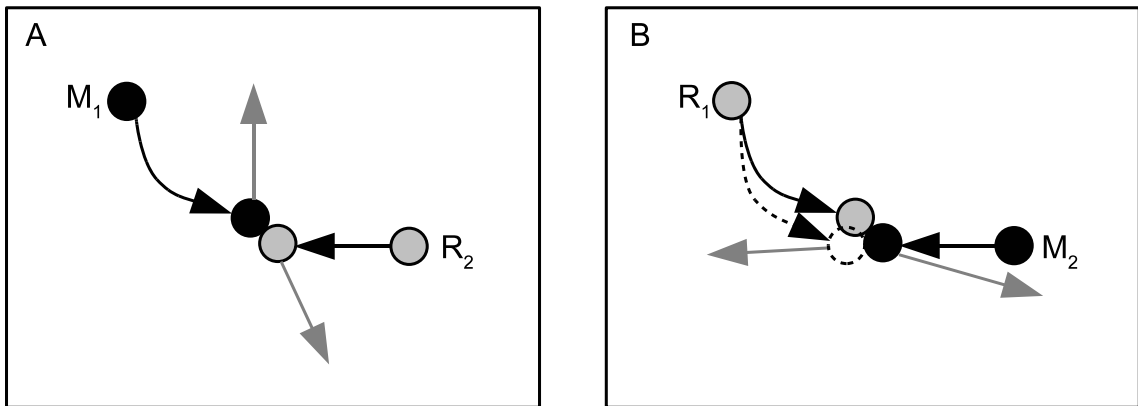
Figure 3.6: Inconsistent collision states causes different trajectories after the collision.

In this situation both A and B detected the same number of collisions, and thus maintain some sense of consistency, but the masters and their replicas have different trajectories after the collision resulting in inconsistent object states. The replica will eventually receive an update from the master and again move in the same trajectory as the master, but the temporary inconsistency can be visually unappealing and

disruptive to game immersion.

3.3 Formalizing Consistency and Correctness

From the discussion above, we can see that in distributed collision detection and resolution, the main challenge is to maintain consistency and correctness among the stations before and after a collision. In this section, we formalize the definition of consistency and correctness for motion prediction, collision detection and collision resolution. Through the formalization, we will generalize the problem for each part and discuss how each part cannot be achieved due to player inputs and extrapolation errors. Finally, we redefine each part in a less constrained way so that some form of consistency and correctness can be efficiently achieved.

3.3.1 Consistency and Correctness in Motion Prediction

The goal of motion prediction in multi-player online games is to predict states for replicas so that they will be consistent with the masters' states. In this section, we want to first formally define state consistency and correctness. Next, we will discuss under what circumstances will a replica have consistent and correct motion prediction.

First we define state consistency:

State Consistency. *A replica R_i^B on B is state consistent with its master M_i on A at time t if and only if the predicted state $\hat{X}_t^{R_i^B}$ of R_i^B is equal to the absolute state X_t of M_i. Two replicas from the same master, R_i^B on B and R_i^C on C, are state consistent with each other at time t if and only if the predicted state $\hat{X}_t^{R_i^B}$ of R_i^B is equal to predicted state $\hat{X}_t^{R_i^C}$ of R_i^C.*

State Correctness. *A replica R_i^B on B is state correct with its master M_i on A at time t if and only if R_i^B is state consistent with M_i.*

By the above definition, state consistency can be between a replica and its master, or two replicas from the same master. When a replica is consistent with its master, it is also correct. In most games, consistency between masters and their replica is

23

more important than consistency between replicas; therefore, we will focus on state correctness in our discussion.

For a replica to achieve state correctness, the master has to be moving at a known (usually zero or constant) acceleration and without any player inputs. This means that the master is moving according to the motion model. The replica can then extrapolate a predicted state without errors. The state of the master is predictable, and the extrapolated state of the replica is accurate.

Replicas are state incorrect if the player enters an input to change the master's motion. The master's motion cannot be predicted, and so the predicted state of the replica is inaccurate and incorrect. Fortunately, the problem of state incorrectness repairs itself. When there are no more player inputs, the master motion is predictable again, and the next state update from the master will allow the replicas to accurately predict a state. Inconsistency may thus be short-term, producing only small deviation errors that are not apparent when rendered. We can relax the criteria of state correctness by allowing replicas to have some extrapolation error if the error is unnoticeable. We can then define the relaxed state correctness as ΔP-state correctness:

ΔP-**State Correctness.** *A replica R_i^B on B is ΔP-state correct with its master M_i on A at time t if and only if the predicted state $\hat{X}_t^{R_i^B}$ of R_i^B and the absolute state X_t of M_i are related such that the difference in positions is bounded: $|\hat{P}_t^{R_i^B} - P_t| \leq \Delta P$.*

The definition of ΔP-state correct is stating that replicas only need to be ΔP close to the master at every frame. The obvious question is how large ΔP should be. When an object is moving in the game without any other objects to collide with, the value of ΔP depends on the game and desired visual result—a relatively large deviation may be acceptable. However, when there are other objects to collide with we want ΔP to be as small as possible to avoid incorrect collision detection. Note that object position is sufficient here, since other aspects of object state such as velocity and acceleration are not required for collision detection and may be inferred from position for resolution.

3.3.2 Consistency and Correctness in Distributed Collision Detection

In a peer-to-peer, multi-player, online game allowing each station to detect and resolve collisions can lead to an inconsistent and incorrect result. Even if the result may eventually be repaired, different stations may fail to visualize an actual collision or represent a collision that did not occur. Certainly missing or extraneous collisions are visually disruptive; collision detection in itself, however, can be important to the design of many games. For example, a bumper car game may use the number of collisions to determines the amount of damage done to a car, calculating the winner as the car with least damage. An inconsistent appearance of collisions misrepresents critical game state to the players.

In this section, we first introduce a way of measuring collision consistency and correctness. Then we will formalize consistency and correctness in collision detection.

Collisions are discrete events associated with time. However, we cannot simply sample a specific time and determine if two stations are consistent in terms of collisions. For example, at t_{10}, A detected a collision between M_1 and R_2 but B did not detect a collision between M_2 and R_1. At t_{20}, there is no collision detected on both stations. If we ask whether A and B are consistent on collision at t_{10}, then the answer is no. If we ask if the stations are consistent at t_{20}, then the answer is yes since neither A nor B detected a collision. It is clear, however, that B missed the collision at t_{10} and is inconsistent semantically, with respect to the number of collisions, even if it is consistent in position.

To evaluate consistency and correctness for collisions, we propose measuring the number of accumulated collisions for each object pair on each station. We denote the collision counter as $c_{i,j}^S$, where i and j are object IDs and S is the station ID. From the above example, each station has a collision counter. At t_{10}, A has $c_{1,2}^A$ of 1 and B has $c_{1,2}^B$ of 0. At t_{20}, the collision counts for both stations remain the same, and so A and B remain inconsistent.

Using the collision counters, we now formally define collision-count consistency:

Collision-Count Consistency. *Two stations, A and B are collision-count consistent for one object pair, O_i and O_j, at time t if and only if the collision counter $c_{i,j}^A$ is equal to the collision counter $c_{i,j}^B$.*

To define collision count correctness, we first define a virtual perfect station:

Virtual Perfect Station. *The virtual perfect station is a hypothetical station that receives all state updates from all stations without network delay or loss. At any given time, the station has correct state for all objects and correct collision count for all object pairs.*

Using the above definition, we define collision count correctness as:

Collision Count Correctness. *A station A is collision count correct for one object pair, O_i and O_j, at time t if and only if the collision counter $c_{i,j}^A$ is equal to the collision counter $c_{i,j}^V$ on the virtual perfect station.*

By the definition above, if two stations are collision count correct on a certain object pair, then they must be collision count consistent. On the other hand, two stations that are collision count consistent on a certain object pair need not to be collision count correct.

For two stations to achieve collision-count consistency, both stations need to detect the same collisions at the same time. However, in practice, at the time of collision the state of the replicas may be different from the masters' state, perhaps ensuring only ΔP-state consistency. The small difference in position can cause one station to detect the collision before or after the other station, but assuming a small enough error bound both stations will eventually agree on the collision. The short interval of inconsistency should be tolerated. We define the interval as the collision inconsistency interval Δt_{col}, and define a relax version of collision count consistency as Δt_{col}-collision count consistency:

Δt_{col}-Collision Count Consistency. *Two stations, A and B are Δt_{col}-collision-count consistent for one object pair, O_i and O_j, if and only if the collision counter $c_{i,j}^A$ at t_A is equal to the collision counter $c_{i,j}^B$ at t_B such that $|t_A - t_B| \leq \Delta t_{col}$.*

Inconsistency in collision-count occurs when the colliding objects are state inconsistent, such that the deviation for the colliding objects are large enough to cause false and missed collisions. This leads to one station detecting more collisions than the other, and Δt_{col} may grow excessive. Unlike state inconsistency, state updates from the masters do not repair an inconsistent collision count. Therefore, one of the goals of this research is to develop efficient algorithms that preventively ensure correct inconsistency in collision counts, and reduce Δt_{col} as much as possible.

As a further extension we note that collision-count *correctness* may not be strictly required, as long as the corresponding collision-count *consistency* is achieved to ensure fairness and eliminate confusion. Two stations may be Δt_{col}-collision-count consistent but incorrect if both stations detect a false collision during Δt_{col}. This has applications to multi-player online games which may tolerate the incorrectness, although more demanding physics simulations may find this unacceptable.

3.3.3 Consistency and Correctness in Distributed Collision Resolution

When stations are allow to resolve collisions independently, if the states of the replica are incorrect at the time of collisions, the calculated post-collision trajectories and states will also be incorrect. As discussed in the above sections, it is difficult to achieve state correctness when there are player inputs, and similarly it is also difficult to achieve state correctness after a collision. In this section, we will formally define post-collision state correctness and a relaxed version.

The definition of post-collision state correctness is directly related to state consistency:

Post-collision State Correctness. *A replica R_i^B on B is post-collision state correct with its master M_i on A after the time of collision t_{col} if and only if R_i^B is state correct with M_i after t_{col} and $\hat{\theta}_i^B$, the direction of R_i^B represented by angles with respect to world frame, and θ_i, the direction of M_i, are equal.*

To define a relaxed version of the above definition, we need to examine what is

27

visually significant for objects after a collision. In state correctness, we point out that the distance between the replica and the master's positions is important for rendering and collision detection. For collision resolution, the direction where the objects bounce off to is also important; therefore, we need a more strict definition for the post-collision state:

$\Delta\theta_{col}$**-post-collision State Correctness.** *A replica R_i^B on B is $\Delta\theta_{col}$-post-collision state correct with its master M_i on A after the time of collision t_{col} if and only if R_i^B is ΔP state correct with M_i after t_{col} and $\hat{\theta}_i^B$, the direction of R_i^B represented by angles with respect to world frame, and θ_i, the direction of M_i, are $\hat{\theta}_i^B - \theta_i \leq \Delta\theta$.*

The definition above states that, after a collision, if the post-collision position of the replica is ΔP close to the master's position, and the post-collision direction of the replica is $\Delta\theta_{col}$ close to the master's direction, then the replica is considered to have a correct state. One of the goals of this research is then to develop algorithms to reduce ΔP and $\Delta\theta_{col}$ as small as possible so that the post-collision states and trajectories are visually acceptable.

Chapter 4
Distributed Collision Detection and Resolution

Collision detection and resolution are essential in modern computer games to display proper visual responses when virtual objects collide. The correct states of these objects are required for the game engine to properly calculate the collision point and time. However, as stated in chapter 3, in multi-player online games, the states of the replicas can deviate from the states of the master due to errors from motion predictions. The collision points and times are then different between the master and its replicas, causing the states to diverge. This leads to inconsistent object states and collision counts among the stations.

In chapter 3, we defined the collision inconsistency interval, Δt_{col}, as the difference in time in detecting collisions between two stations. When two stations both detect the same collision but at different time, the stations are Δt_{col}-collision count consistent. However, if one station misses a collision, Δt_{col} will then grow unbounded. Thus, our first goal is to design a protocol that bounds the collision inconsistency interval, so that stations will remain Δt_{col}-collision count consistent even if one station misses a collision.

Just bounding the collision inconsistency interval is not enough to produce good visual results. When Δt_{col} is large, it indirectly means that the states of the objects at the time of collision between the stations are very different. Different collision

states will then produce different post-collision trajectories, and $\Delta\theta_{col}$ will be large. Large $\Delta\theta_{col}$ causes the replicas to move in different directions from the masters after collisions, and the states diverge further. Furthermore, upon receiving the next state update from the master, the replicas correct their position with a large jump. This large discrepancy is visually confusing. Therefore, our next goal is to minimize Δt_{col} and $\Delta\theta_{col}$ so that stations detect and resolve the collisions as close as possible to reduce visual confusion.

With the above goals in mind, in this chapter, we will discuss various algorithms and protocols in dealing with the problems in distributed collision detection and resolution. First, we categorize the different types of objects that can exist in a multiplayer online game, and the types of collisions between them. We then focus our discussion on master-replica collisions and define two categories of stations that are involved in a master-replica collision.

Next, we present two collision detection agreement protocols that bound and minimize the collision inconsistency time. These two protocols are novel ideas in solving the problems; however, both have flaws. The post-collision protocol bounds Δt_{col} but does not minimize it. On the other extreme, the pre-collision protocol tries to have a zero collision inconsistency interval, but may still have unbounded Δt_{col}. We present these two protocols to help understand the complexity of the problems, and motivate the design of our third protocol, the motion-lock protocol.

The motion-lock protocol bounds and minimizes the collision inconsistency interval by locking the motion of the colliding objects for a short period of time. The locking of motion temporarily discards (or buffers) all a player's commands; this may reduce the responsiveness of the objects, but allows the objects to commit to the collision ahead of time. In our discussion we will show that with minimal amount of locking the objects can achieve short Δt_{col}. Furthermore, the motion-lock protocol allows the post-collision trajectory to be calculated before the collision, and sent to the other stations to achieve post-collision trajectory agreement, minimizing $\Delta\theta_{col}$.

Finally, we discuss the problems associated with collisions with multiple objects. The motion locking blocks player controls to allow the stations to notify each other before the collision. However, when there are multiple objects in the scene, objects

that are motion locked and committed to the collision can be interrupted by other objects. The previous committed collision is no longer valid, and the motion-lock protocol is no longer correct. We introduce the Spatial-temporal Bucket Synchronization algorithm that resolves a group of colliding objects at the same time.

4.1   Categorization of Collisions and Stations

In offline games, all objects are local to the game process. Collisions are calculated by single process on the station. However, in multiplayer online games, different types of objects exist on the station, and so stations require protocols to resolve the collisions.

Before we discuss the collision detection protocols, we first need to identify the different types of objects that can exist on a station. There are three types of objects that can exist on a station. From the previous chapters, we have already identified the masters and replicas. The masters are objects that are local to the stations. Players or the local game processes can control the masters. The replicas are copies of the masters that reside remotely on other stations. The local station sends the state of the masters through the network to update the replicas.

The third type of object is the global object. Global objects are objects that are controlled by the game process itself. They can be static, such as the world boundaries and obstacles, or dynamic, such as the non-player characters (NPCs). Static global objects are stateless, and so they are the same on all stations. For dynamic global objects, since they are controlled by the game process, if the object states of two instances diverge, the states can be easily predicted and corrected by the game process because no player inputs are involved.

If we assume that each player controls only one master, from the three types of objects, there exist five categories of collision: master-global, master-replica, replica-global, replica-replica, and global-global (if players can control more than one object there will be master-master collisions). Since global objects appear on all stations, they are considered local on all stations; therefore, master-global and global-global

collisions are resolved by the game process. For replica-global and replica-replica collisions, the stations that detected these collisions have no authority over the replicas; therefore, whether these replicas collided or not will be determine by their masters' station.

Master-replica collisions are the main source of inconsistency we will consider in distributed collision detection. Before we discuss the master-replica collisions, we first define the stations that are involved in master-replica collisions as the collision participating stations.

Collision Participating Stations. *For each master-replica collision, there are exactly two participating stations involved. When one station A detects a collision between its master M_i and a replica R_j^A, there exists a station B such that B detects a collision between its master M_j and a replica R_i^B, where R_i^B is a replica of M_i and R_j^A is a replica of M_j.*

In a master-replica collision, the collision participating stations have authority over the detection and resolution of the collision. Since both of the collision participating stations have the authority, however, they need to communicate and agree on the collision. For the rest of this chapter, we will focus on the agreement between the collision participating stations.

4.2 Distributed Collision Detection

In a master-replica collision, the two participating stations may detect the collision at different times; we term the time difference as the collision inconsistency interval Δt_{col}. The collision inconsistency interval may be infinite if one of the participating stations missed the collision. To avoid this, the other participating station needs to notify the station about the collision so that the interval will be bounded, and the stations will be collision count consistent.

The notification about the collision is sent after the collision with the receive time dependent on the network latency. Therefore, to reduce the collision inconsistency interval, the notification should be sent before the collision. This will also reduce the

divergence of the object states after collisions and improve the visual play-out of the collision.

In this section we present three collision detection agreement protocols that deal with bounding and minimizing the collision inconsistency interval for master-replica collisions. The first protocol is the *Post-Collision Protocol* that bounds the collision inconsistency interval. The stations naively send out collision counts after collisions to prevent the collision inconsistency interval to grow to infinity. Furthermore, heart-beat messages are sent to provide on average collision count consistency within a bounded time.

The second protocol is the *Pre-Collision Protocol* that tries to make the stations come to an agreement on collisions before the objects collide. The stations predict collisions based on objects' motions, and exchange messages before the collision. If the stations reach an agreement, the collision inconsistency time will be zero; however, as we shall see, in an asynchronous network, reaching an agreement before the collision time is quite difficult.

The third protocol is the *Motion-Lock Protocol* that combines the above two protocols to provide collision count consistency with reduced collision inconsistency interval. When the stations predict a collision, the motions of the objects are locked to commit to the collision. The motion locking allows the collision count to be incremented and sent before the collision, and thus reduces the length of the collision inconsistency interval.

In the next three sections, we present the protocols. For all protocols, we assume the network communication is asynchronous and unreliable. Packets can be delayed for arbitrary time and may be lost in the network. We do assume a practical context where messages will eventually get though with enough resends.

4.2.1 Post-Collision Agreement Protocol

The Post-Collision protocol maintains consistency through exchanging collision counts between the stations. When a station detects a collision, it first increments the collision counter, and then sends the counted value to the other participating station.

If the receiving station missed the collision, the local collision count will be less than the received collision count. This triggers the collision event, and the station will resolve the collision and increment the local collision counter to reach collision count consistency.

When the network latency is high, the collision time for the received counter may be very out-dated. Calculating long delayed collisions can produce confusing visual result. Therefore, received counters that exceed a preset cut-off time will still increment the local collision counter but the post-collision trajectory is not calculated. Furthermore, because packets can get lost in the network, the collision counts that are sent after each collision may not reach the receiving station. The protocol then needs to send the collision counts at a heart-beat interval to bound the collision inconsistency interval.

Formally, we assume the following. Let:

- A and B be the participating stations.

- M_i be the master on A, and R_i^B be the replica of M_i on B.

- M_j be the master on B, and R_j^A be the replica of M_j on A.

- $CollisionDetection(obj_1, obj_2)$ be a function that returns true if obj_1 and obj_2 have collided, and returns false otherwise.

- $CollisionResolution(obj_1, obj_2)$ be a function that calculates the post-collision trajectories for obj_1 and obj_2.

- $DelayedCollisionResolution(obj_1, obj_2)$ be a function that handles delayed trajectory calculation for delayed collisions.

- On A, $c_{i,j}^A$ be the local collision counter for collisions between M_i and R_j, and $t_{i,j}^A$ be the time of collision. $c_{i,j}^B$ be the received collision counter from B to A, and $t_{i,j}^B$ be the received collision time.

- Station B has the same counters and timestamps with reversed station id.

- T_{cut} be the cut-off time for resolving delayed collision events.

- T_{beat} be the preset heart beat interval, and $ts_{i,j}^A$ be the last time $c_{i,j}^A$ was sent.

Algorithm 1 shows the post-collision protocol.

The protocol starts when A detects a collision between M_i and R_j^A, A increments the collision counter $c_{i,j}^A$, calculates the post-collision trajectories, and then sends the collision counter to B. If A receives $c_{i,j}^B$, A compares its local collision counter, $c_{i,j}^A$, with the received counter. If A missed a collision, $c_{i,j}^B$ will be greater than $c_{i,j}^A$. This triggers A to resolve the collision and increments the counter. Collision events that are triggered by the collision counters are usually past the actual collision time. These collisions need to be resolved differently to properly calculate the post-collision trajectory. This will be discussed in a later section of this chapter.

Evaluation of Post-Collision Agreement Protocol

For the Post-Collision protocol, because collision counters are sent after collisions have been detected, and network latency means that the station which missed the collision may trigger the collision event after the objects have move passed the actual collision point. The length of the collision inconsistency interval is then the length of network latency, and so the time performance of the protocol depends on the network condition. Figure 4.1 depicts the collision inconsistency interval between the two stations.

Furthermore, objects on the station that delay the collision event may appear to bounce off each other with a gap in between. To prevent large gaps, the cut-off time discards collision counters with long delay. From the research of Pantel et al. [23], delays of events under 200ms can be tolerated by players. Therefore, for our current implementation, we set T_{cut} to 200ms. However, bad network condition can increase the average latency over 200ms. Most of collisions in this case will be discarded.

Another disadvantage of this protocol is that it can lead to incorrect collision count. When one collision participating station detects a false collision, it will send the collision count and the stations will agree on the false collision. The stations

Algorithm 1 Post-Collision Agreement Protocol

On initialization at A

    for each replica R_j on A **do**

        $c_{i,j}^A \leftarrow 0$

        $t_{i,j}^A \leftarrow 0$

        $ts_{i,j}^A \leftarrow 0$

        $c_{i,j}^B \leftarrow 0$

        $t_{i,j}^B \leftarrow 0$

    end for

On each game loop at A

    for each replica R_j on A **do**

        if $CollisionDetection(M_i, R_j) = true$ **then**

            $CollisionResolution(M_i, R_j)$

            $c_{i,j}^A \leftarrow c_{i,j}^A + 1$

            $t_{i,j}^A \leftarrow t_{current}$

            $ts_{i,j}^A \leftarrow t_{current}$

            send $< c_{i,j}^A, t_{i,j}^A >$ to B

        else if $c_{i,j}^B > c_{i,j}^A$ **then**   ▷ Compare collision counts.

            if $t_{current} - t_{i,j}^B < T_{cut}$ **then**

                $DelayedCollisionResolution(M_i, R_j)$

            end if

            $c_{i,j}^A \leftarrow c_{i,j}^B$

            $t_{i,j}^A \leftarrow t_{current}$

        end if

$\quad$ **if** $t_{current} - ts_{i,j}^A > T_{beat}$ **then**

$\qquad ts_{i,j}^A \leftarrow t_{current}$

$\qquad$ send $< c_{i,j}^A, t_{i,j}^A >$ to B

$\quad$ **end if**

end for

On receipt of $< c_{i,j}^B, t_{i,j}^B >$ *from* B

$\quad$ store $c_{i,j}^B$ and $t_{i,j}^B$ at A
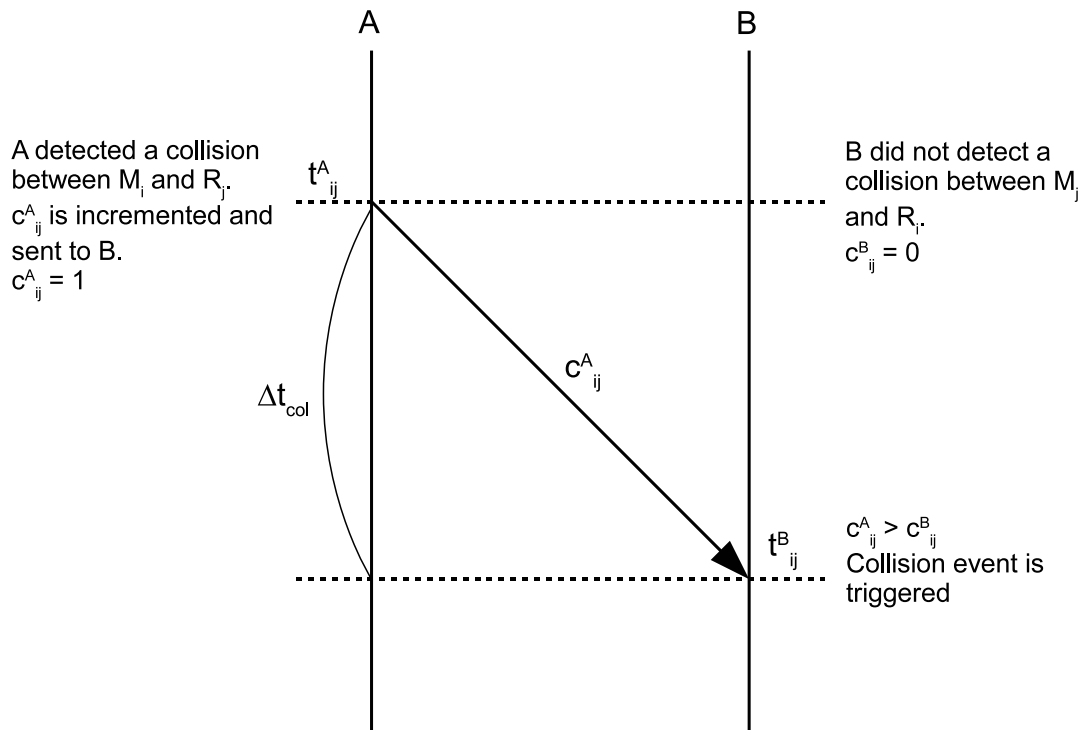
Figure 4.1: Post-collision agreement protocol. Length of Δt_{col} depends on the network latency.

are consistent but incorrect. This means the protocol's correctness depends on the state correctness of the replicas. Therefore, the station that suffers bad network

37

communication and detects many false collisions will be the station that others agree upon.

The heart-beat intervals must be infrequent to prevent any excessive bandwidth cost. Since counters that have arrived later than the cut-off time will be discarded, the purpose of heat-beat messages is to loosely synchronize collision counters. The value of T_{beat} should be tuned depending on the game requirements.

4.2.2   Pre-Collision Agreement Protocol

The goal of pre-collision agreement is to have the participating stations reach an agreement on a master-replica collision before the objects collide. If an agreement can be reached, the collision inconsistency interval can be as low as zero. In order to achieve this, the participating stations need to exchange messages before the collision.

In this section we present the Pre-Collision agreement protocol. Similar to the classic 2-phase commit protocol, the pre-collision agreement protocol consists of two phases: the voting phase and the collision phase. In the voting phase, the two collision participating stations exchange votes on a potential collision. The motions of the objects are locked so that the votes do not change. In the collision phase, if both stations agreed on the collision, the objects are committed to collide and the collision counters are incremented. If one station disagrees on the collision, or the stations are unable to complete the protocol before the collision time, both stations abort the collision.

The Pre-Collision agreement protocol requires at most 3 rounds of message exchange to reach an agreement. This introduces round-trip delays similar to the client/server architecture. However, in the client/server architecture, the centralized server handles all collision messages from all the clients and requires large bandwidth usage. On the other hand, in the Pre-Collision protocol, for each collision, only the two collision participating stations are involved with the message exchange, and thus the bandwidth requirements are distributed.

Formally, in addition to the assumptions we made in the Post-Collision protocol, we further assume the following. Let:

- $CollisionPrediction(obj_1, obj_2)$ be a function that returns true if the current states of obj_1 and obj_2 will lead to a collision, and returns false otherwise.

- $LockMotion(obj)$ and $UnlockMotion(obj)$ be functions that locks and unlocks the motion of obj.

- $willCollide[k]$ be a Boolean array that indicates if the master of the station will collide with replica k.

- $canCollide[k]$ be a Boolean array that indicates if the participating stations have agreed on collision between the master and replica k.

Algorithm 2 shows the pseudocode of the pre-collision agreement protocol.

On each game loop, all stations check for potential collisions for each object pair using the collision prediction algorithm. The collision prediction algorithm first estimates an object's future trajectory by linear extrapolation of the current state. The estimated trajectories are checked for intersections. If the trajectories intersect, the pair of objects will collide at a future time if the objects maintain their velocities.

When a station detects a potential collision, the station initiates the voting phase of the protocol. If station A predicted a collision between M_i and R_j, the motion of M_i and R_j are locked such that player inputs and master updates do not change M_i's and R_j's motion, and the objects are guaranteed to collide. Without locking, no agreement can ever be reached. After the motions are locked a request, REQ, is sent to B to inform B of the incoming collision. $willCollide[j]$ is then set to true to indicate that the objects have their motions locked and the station is waiting for a vote.

Next, if A receives a REQ from B, A checks if it has also detected a potential collision. If no potential collision is detected, A sends a NO vote to inform B that the collision will lead to a false collision and should be aborted. If A has also detected a potential collision and has its objects' motion locked, A sends a YES vote to B to engage the collision.

In the collision phase, when A receives a YES vote from B, A sends an ACK acknowledgement to B to confirm the collision. A sets $canCollide[j]$ to true and waits

Algorithm 2 Pre-Collision Agreement Protocol

On initialization at A

    for each replica R_j on A **do**

        $c_{i,j}^A \leftarrow 0$

        $c_{i,j}^B \leftarrow 0$

        $willCollide[j] \leftarrow false$

        $canCollide[j] \leftarrow false$

    end for

On each game loop at A

    for each replica R_j on A **do**

        if $CollisionPrediction(M_i, R_j) = true$ **then**

            $LockMotion(M_i)$

            $LockMotion(R_j)$

            $willCollide[j] \leftarrow true$

            send REQ to B ▷ Send a vote request to initate the protocol.

        end if

        if $CollisionDetection(M_i, R_j) = true$ **then**

            if $canCollide[j] = true$ **then**   ▷ Commit to the collision.

                $CollisionResolution(M_i, R_j)$

                $c_{i,j}^i \leftarrow c_{i,j}^i + 1$

            end if

            $UnlockMotion(M_i)$

            $UnlockMotion(R_j)$

            $willCollide[j] \leftarrow false$

            $canCollide[j] \leftarrow false$

        end if

    end for

On receipt of a REQ from B at A
    if $willCollide[j] = true$ **then**   ▷ Return a vote when a request is received.
        send YES to B
    else
        send NO to B
    end if

On receipt of a YES from B at A
    if $willCollide[j] = true$ **then**   ▷ Send a ACK to confirm the YES vote.
        $canCollide[j] \leftarrow true$
        send ACK to B
    end if

On receipt of a NO from B at A
    $UnlockMotion(M_i)$ ▷ Abort the collision when a NO vote is received.
    $UnlockMotion(R_j)$
    $willCollide[j] \leftarrow false$
    $canCollide[j] \leftarrow false$

On receipt of a ACK from B at A
    if $willCollide[j] = true$ **then**
        $canCollide[j] \leftarrow true$ ▷ Commit to the collision when a ACK is received.
    end if

---

for the objects to collide. If a *NO* vote is received, A sets $canCollide[j]$ to false and unlocks the motions of the object pair. When A receives an acknowledgement ACK from B, A sets $canCollide[j]$ to true and waits for the objects to collide.

At the time of the collision between M_i and R_j, if $canCollide[j]$ is true, the collision counter $c_{i,j}^A$ is incremented and the post-collision trajectories are calculated. If $canCollide[j]$ is false the collision is aborted. In both cases the motions of M_i and R_j are unlocked afterwards.

Finally, if the protocol is not able to complete by the time the objects collide, the collision is aborted because the stations failed to come to an agreement.

The protocol only requires the collision participating stations to exchange messages, so both stations can initiate the first phase and the protocol will still come to the same result. Figure 4.2 shows the exchange of messages between the collision participating stations. Stations A and B agreed on the collision and the protocol is able to be complete before the collision time.

Evaluation of the Pre-Collision Agreement Protocol

In this section, we evaluate and discuss the problems of the above pre-collision agreement protocol.

By default, if the protocol terminates during the first phase, both stations abort the collision. We designed the protocol to be pessimistic so that in case of a failure during first phase, the stations will still remain consistent.

However, there are two major problems that can cause the Pre-Collision protocol to fail and lead to inconsistent collision counts. The protocol produces inconsistency at the second phase when one of the participating stations ends the protocol before it receives the acknowledgement message. The station that sends the acknowledgement commits to the collision, while the station that did not receive the acknowledgement aborts the collision. The protocol can end during the second phase due two factors: the asynchronous network and the object dynamics.

In an asynchronous network, messages can be delayed and arrive at arbitrary time. If the ACK is received on time, as in figure 4.2, both stations will allow the

A detected a potential collision between M_i and R_j.
Motion of M_i and R_j are locked.

A

B

t_{start}

REQ

B agrees with the collision request.
YES is sent.
Motion of M_j and R_i are locked.

YES

Collision agreed.
ACK is sent.
Wait for collision.

ACK

Collision agreed.
Wait for collision.

Objects on both stations collided at t_{col}
$\Delta t_{col} = 0$.
Motion of the objects are unlocked.

t_{col}
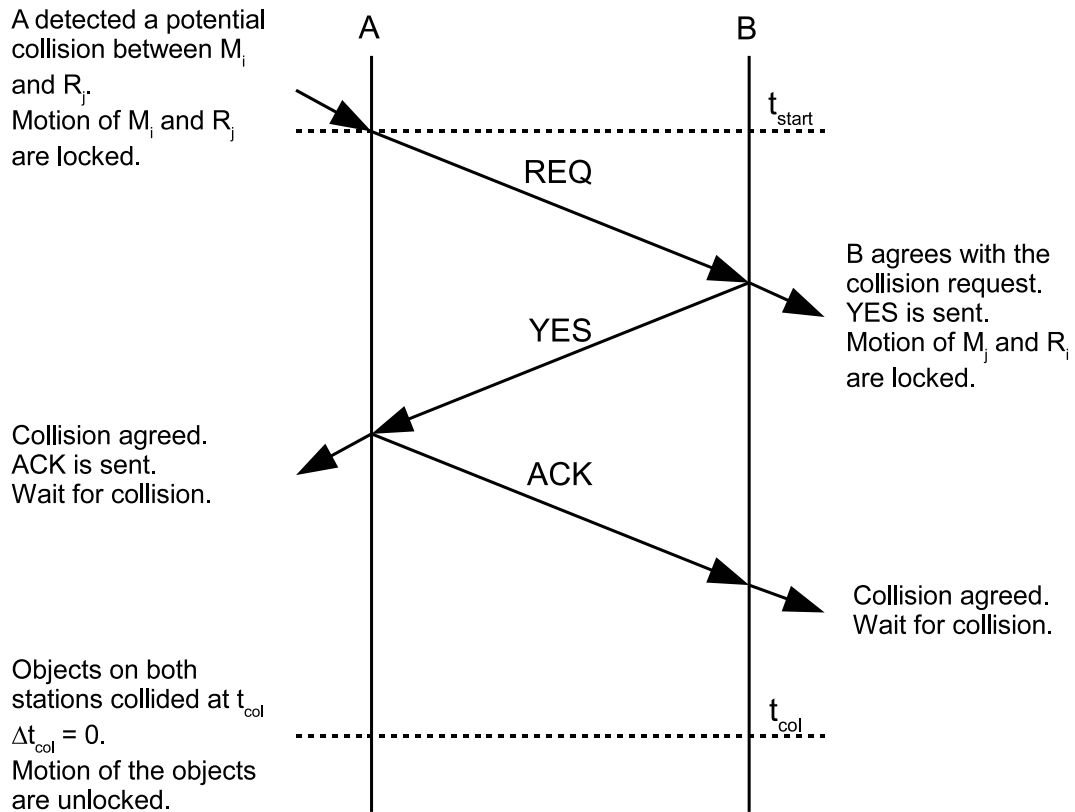
Figure 4.2: Pre-Collision Agreement Protocol message passing.

collision and the collision counts will be consistent. However, if the ACK arrives late and passes the collision time, one station will commit to the collision while the other aborts the collision. Even if the protocol starts early, we cannot guarantee that the ACK message will arrive on time given the none-deterministic nature of the asynchronous network. Figure 4.3 shows the ACK message arriving after the collision time.

The dynamics of the objects can also cause the protocol to fail and lead to inconsistency. For example, if the network has in average 100 ms latency. The protocol requires at least 3 messages to allow a collision, and so approximately 300 ms is required to complete the protocol. If a potential collision is detected such that the objects will collide within 250 ms, the protocol is destined to terminated while the
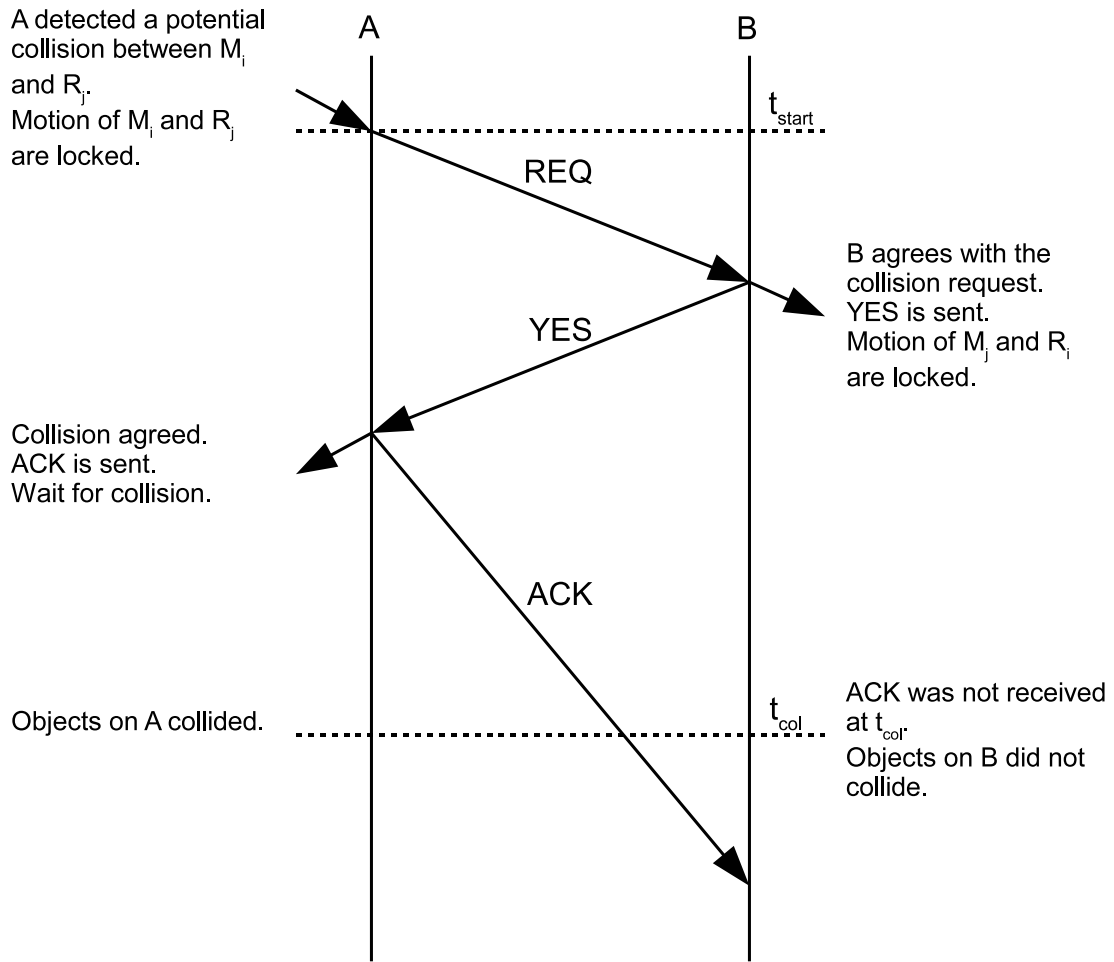
Figure 4.3: Termination of the protocol during passing of ACK causes inconsistency.

ACK is in transit. This will lead to inconsistency. Figure 4.4 depicts the above problem. The dynamics of the objects do not give enough time for the protocol to finish.

Furthermore, if the objects move around frequently, the protocol will rarely be complete. The objects will not collide most of the time and appears to pass through each other. If the game requires objects to collide to determine a winner, the game can become frustrating and visually confusing. The locking of object motions also reduces responsiveness of the objects. If there are many objects in the game, many collisions can happen and the motions of the objects are locked most of the time. The

44

A detected a potential collision when the objects are close to each other. The start time is close to the collision time.

t_{start}

REQ

B agrees with the collision request.

YES

Objects on A collided.

ACK

t_{col}

The protocol did not have enough time to finish. B did not receive ACK in time.
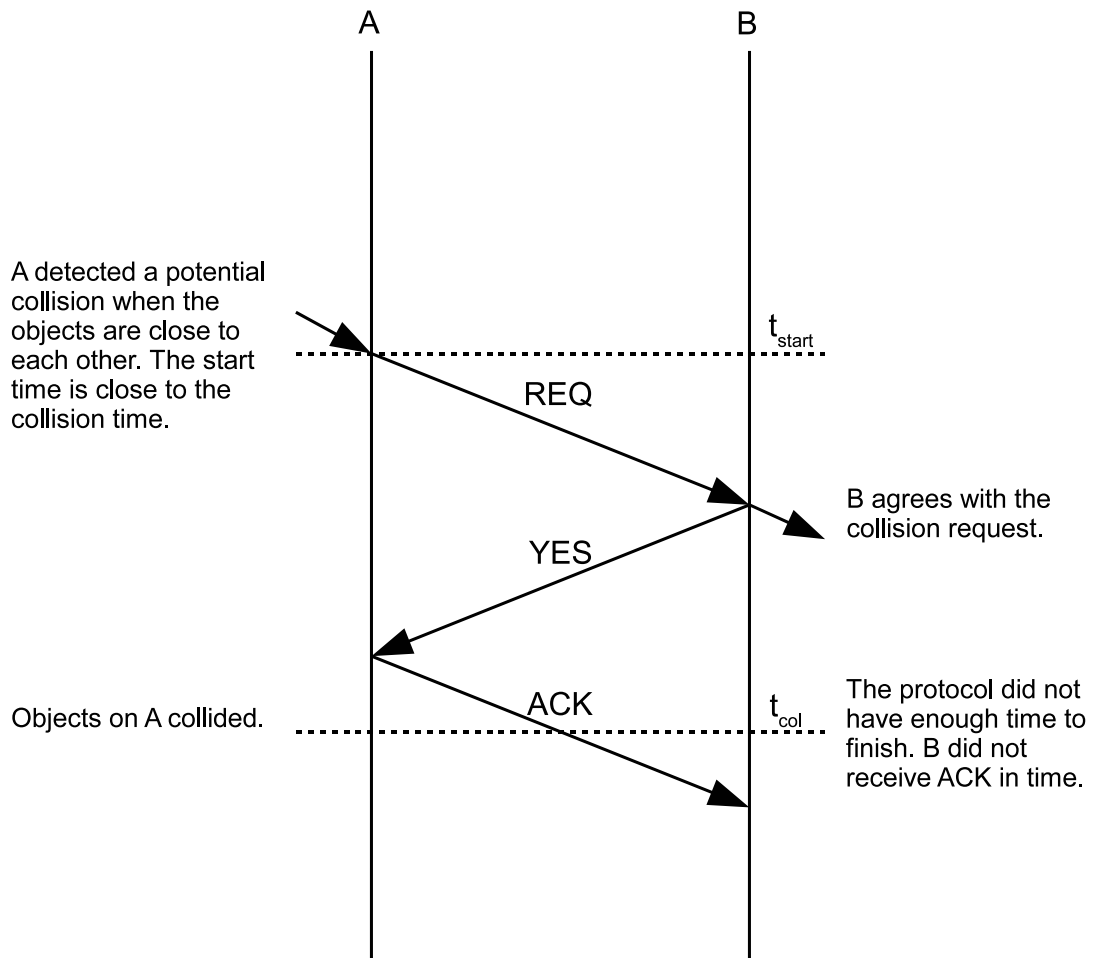
Figure 4.4: Unable to complete the protocol due to late detection of a potential collision.

players cannot control their characters and the game becomes unplayable.

In general, in an asynchronous network, guaranteeing an agreement between stations within a given length of time is difficult [18]. Therefore, the Pre-Collision protocol is unable to maintain collision count consistency. However, if both phases of message exchange can be finished, the protocol can achieve zero collision inconsistent intervals. This protocol may be ideal for games running on more reliable and low-latency network.

4.2.3 Motion-Lock Collision Agreement Protocol

From the above sections, we see that the Pre-Collision protocol does not provide collision count consistency in every case; however, if it is able to complete, it can achieve collision count consistency with Δt_{col} equals to zero. On the other hand, the Post-Collision protocol provides on average collision count consistency, but the length of Δt_{col} depends on the network latency. In this section, we present the Motion-Lock protocol that provides on average collision count consistency with Δt_{col} less than the network latency. This is done by combining the techniques of motion locking in the pre-collision protocol and collision count exchange in the post-collision protocol.

The Motion-Lock collision agreement protocol is similar to the post-collision protocol such that stations exchange collision counts to maintain consistency. To reduce the length of inconsistency time, stations lock the motions of the colliding objects so that they are committed to the collision. This allows the collision counters to be sent early before the collision.

However, if the objects' motions are locked for too long, players may feel they have lost control of the objects. To prevent this, a predefined maximum locking threshold T_{lock} caps the locking interval. When two objects' estimated trajectories intersect, their motions are locked only when the remaining time to the collision is between T_{lock} and 0.

Formally, in addition to the assumptions we made in the above two protocols, we further assume the following. Let:

- $CollisionPrediction(obj_1, obj_2)$ be a function that returns the remaining time to a collision if the current states of obj_1 and obj_2 will lead to a collision, and returns ∞ otherwise.

- On A, $cp_{i,j}^A$ be the local potential collision counter for the potential collisions between M_i and R_j, and $cp_{i,j}^B$ be the received potential collision counter from B to A.

- T_{lock} be the preset maximum locking threshold.

Algorithm 3 Motion-Lock Protocol

On initialization at A

    for each replica R_j on A **do**

        $c_{i,j}^A \leftarrow 0$

        $cp_{i,j}^A \leftarrow 0$

        $t_{i,j}^A \leftarrow \infty$

        $ts_{i,j}^A \leftarrow 0$

        $c_{i,j}^B \leftarrow 0$

        $cp_{i,j}^B \leftarrow 0$

        $t_{i,j}^B \leftarrow \infty$

    end for

On each game loop at A

    for each replica R_j on A **do**

        $\Delta t_{remain} \leftarrow CollisionPrediction(M_i, R_j)$

        if $0 \leq \Delta t_{remain} \leq T_{lock}$ **then**

                $\triangleright$ Lock motions and exchange potential collision counters.

        $LockMotion(M_i)$

        $LockMotion(R_j)$

        $t_{i,j}^A \leftarrow \Delta t_{remain} + t_{current}$

        $cp_{i,j}^A \leftarrow cp_{i,j}^A + 1$

        $ts_{i,j}^A \leftarrow t_{current}$

        send $< cp_{i,j}^A, t_{i,j}^A >$ to B

        else

        $UnlockMotion(M_i)$

        $UnlockMotion(R_j)$

        end if

if $CollisionDetection(M_i, R_j) = true$ **then**

    $CollisionResolution(M_i, R_j)$

    $c_{i,j}^A \leftarrow c_{i,j}^A + 1$

    $t_{i,j}^A \leftarrow t_{current}$

    $ts_{i,j}^A \leftarrow t_{current}$

    send $< c_{i,j}^A, t_{i,j}^A >$ to B

else if $cp_{i,j}^B > cp_{i,j}^A$ **and** $t_{i,j}^B \leq t_{current}$ **then**

                    ▷ Potetial collision counters trigger the collision event.

    if $t_{current} - t_{i,j}^B < T_{cut}$ **then**

        $DelayedCollisionResolution(M_i, R_j)$

    end if

    $cp_{i,j}^A \leftarrow cp_{i,j}^B$

    $c_{i,j}^A \leftarrow cp_{i,j}^B$

    $t_{i,j}^A \leftarrow t_{current}$

    $t_{i,j}^B \leftarrow \infty$

else if $c_{i,j}^B > c_{i,j}^A$ **then**   ▷ Compare collision counts.

    if $t_{current} - t_{i,j}^B < T_{cut}$ **then**

        $DelayedCollisionResolution(M_i, R_j)$

    end if

    $c_{i,j}^A \leftarrow c_{i,j}^B$

    $cp_{i,j}^A \leftarrow c_{i,j}^B$

    $t_{i,j}^A \leftarrow t_{current}$

end if

if $t_{current} - ts_{i,j}^A > T_{beat}$ **then**

    $ts_{i,j}^A \leftarrow t_{current}$

    send $< c_{i,j}^A, t_{i,j}^A >$ to B

end if

end for

On receipt of $a < cp_{i,j}^B, t_{i,j}^B >$ from B

    store $cp_{i,j}^B$ and $t_{i,j}^B$ at A

On receipt of $< c_{i,j}^B, t_{i,j}^B >$ from B

    store $c_{i,j}^B$ and $t_{i,j}^B$ at A

Algorithm 3 shows the pseudocode for the Motion-Lock protocol.

A station initiates the protocol when the collision prediction algorithm detects a potential collision between two objects. The collision prediction algorithm returns the remaining time to the collision. If the remaining time is between the predefined T_{lock} and 0, the motions of the objects are locked so that the objects will eventually collide. Once the objects' motions are locked they are committed to the collision. The potential collision counter can then be incremented and sent to inform the other stations before the objects actually collide. The collision event can be triggered by the object collision itself, collision counters, and the potential collision counters. Because the potential collision counters are sent early, they can be received before the scheduled collision time. To prevent the station triggering the collision event ahead of time, the collision time is also sent so that the stations can correctly schedule the collision event.

Evaluation of Motion-Lock Protocol

The Motion-Lock protocol provides Δt_{col}-collision-count consistency and reduces the collision inconsistent interval by locking the objects' motion and exchanging the potential collision counts before the actual collision. The value of T_{lock} may depend on the game; however, in general, the value should not be too long and perceivable by the players. From the research of Pantel et al. [23], delays of player inputs become perceivable above 100ms; we thus set T_{lock} to 100ms.

The protocol separates the collision counters into a potential collision counter and the actual collision counter. The purpose of doing this is so that the actual collision

counter is not incremented before the collision, and other game events that rely on the collision counter will not be triggered prematurely.

The minimization of the collision inconsistency interval depends on how early the potential collision counters are sent, which itself depends on the dynamics of the objects. The best case is when the potential collisions are detected at T_{lock}, and so the collision inconsistency interval is equal to $networklatency - T_{lock}$. The worst case is that the potential collision is never detected due to frequent changes in objects' motion. The potential collision counter is then sent at the collision time, and so the collision inconsistency interval is equal to the network latency; and thus, the time performance for the Motion-Lock protocol would be better, but never worse, than the post-collision protocol. Figure 4.5 shows the reduced Δt_{col} as the result of passing $cp_{i,j}^A$ before the time of collision. $c_{i,j}^A$ and $c_{i,j}^B$ are incremented afterwards.

Similar to the Post-Collision protocol, the Motion-Lock protocol does not filter out the false collisions. Moreover, when objects are close to each other, both stations force the objects to collide by locking their motions, making it is hard to define collision correctness. If thresholds are too large the objects may appear to be pulled toward each other in order to collide. Therefore, the size of T_{lock} is again important and needs to be scaled to the game object size and speed.

4.3 Distributed Collision Resolution

In the Post-Collision protocol and in the worst-case of the Motion-Lock protocol, stations that missed the collision receive the collision counter after the objects have moved past the collision point. If the collision time is within the cut-off time, the stations need to resolve the collision. However, since the objects are no longer at the collision point, using their current state to calculate collision resolution will result in incorrect post-collision trajectories. The objects on the participating stations will have different post-collision trajectories, resulting in further divergence of the states between the replicas and their masters.

One way of calculating collision resolution is to determine the collision normal.
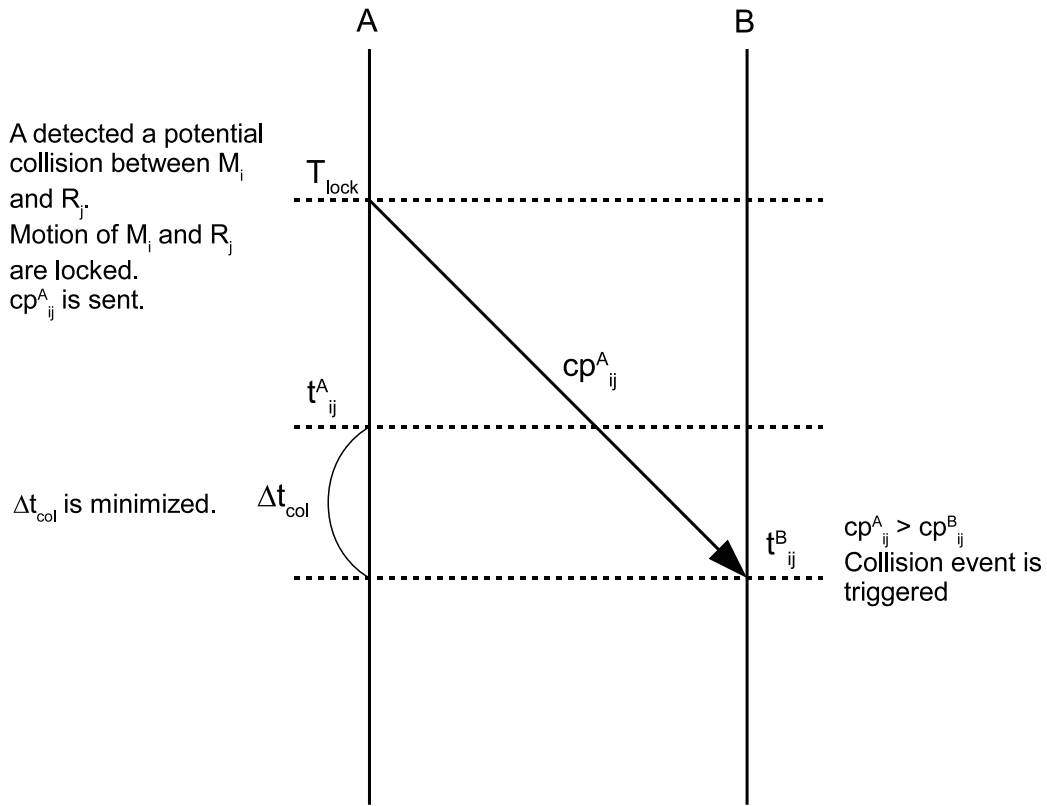
Figure 4.5: By sending potential collision counter before the collision, Δt_{col} can be minimized.

Figure 4.6 shows the collision between two spheres. Here the collision normal can be calculated by finding the vector between the centers of the spheres. The direction of the post-collision trajectory can then be determined by inverting the direction of the pre-collision trajectory along the collision normal. However, in figure 4.7, as the objects move apart, the collision normal can no longer be calculated using the sphere center. The post-collision trajectories would be completely different than on the other stations if the above calculation of the collision normal is used. Therefore, collision resolution for the delayed collisions needs to be calculated differently.

The problem is of course magnified over time. After the replicas travel with their post-collision trajectories for a while, the next state update from the master will result in a correction, jumping (or converging) from the current position to the update
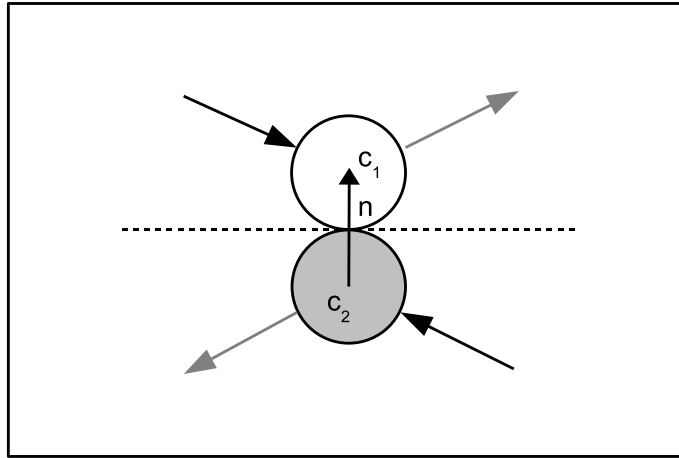
Figure 4.6: Collision with correct collision normal.
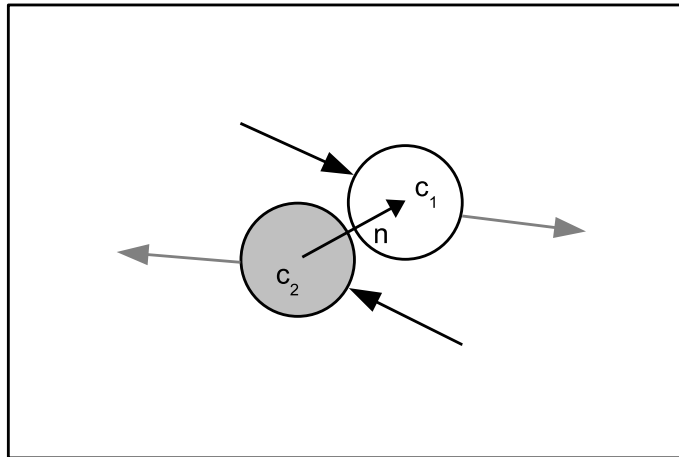
Figure 4.7: Collision with incorrect collision normal.

position. If the post-collision trajectory is incorrect and $\Delta\theta_{col}$ is large, the correction can be large and visually confusing. Figure 4.8 shows a large discrepancy in post-collision trajectories and the required large correction. If there are many collisions, frequent corrections can reduce playability. If smooth convergence is applied to correct the position instead of jumps, the large smoothing trajectory can also cause more false collisions.

Therefore, in distributed collision resolution, an important goal is to coordinate the stations so that correct post-collision trajectories are used to avoid further divergence.
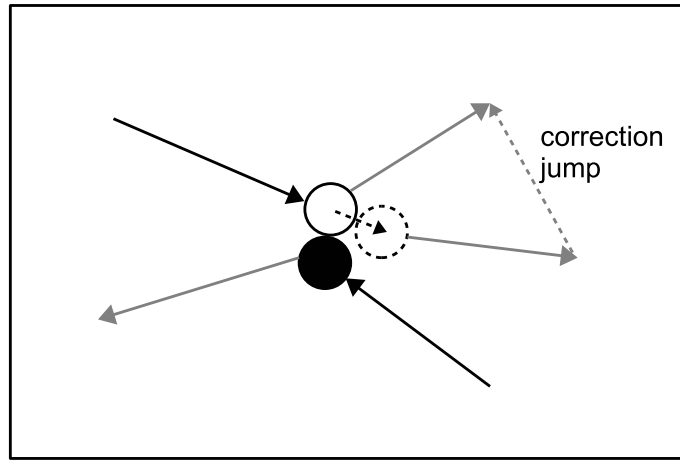
Figure 4.8: Large correction jump when state update is received.

4.3.1 Post-Collision Trajectory Agreement

To deal with the above problems we modify our Motion-Lock protocol. When objects are committed to a collision and have locked their motions, they will travel linearly until the collision. It is possible then to accurately calculate the post-collision trajectory before the actual collision by linear extrapolation of the current state to the collision time, and then calculating the post-collision trajectories. Along with the collision counter and collision time we can then send the pre-calculated post-collision trajectories to the other participating stations. Upon receiving the post-collision trajectory, if the station missed the collision, the received post-collision trajectories can be used to update the objects' states. Objects on both the participating stations will then travel in the same direction after the collision.

The position of the object at the collision point can also be pre-calculated with linear extrapolation, and be sent before the collision. However, as mentioned above, objects on the stations that missed the collision have moved past the collision point. Objects would then require warping back to the received collision point, and such warping can cause visual confusion.

With the received post-collision trajectories, the stations are applying the trajectories to the objects at different positions. The size of ΔP would depend on when the

post-collision trajectories are received. Fortunately, the Motion-Lock protocol minimizes the collision inconsistency interval, and so the deviations of the replicas are not so far off from the master, and ΔP may be small and not perceivable. Furthermore, since the received post-collision trajectories are used, the direction of objects on both stations are equal such that $\Delta\theta_{col}$ is zero. Starting positions of the post-collision trajectories may be slightly different, but the directions are the same. This minimizes the correction jump when the next state update is received. Figure 4.9 shows the post-collision trajectories with different starting points, and the reduced correction jump.
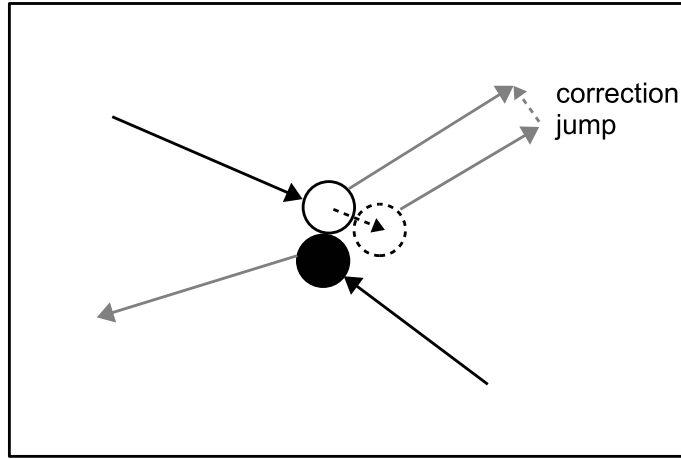
Figure 4.9: Small correction jump using received post-collision trajectory.

The above post-collision trajectories agreement strategy benefits the stations that missed the collision. However, if a station received the post-collision trajectories and also detected the collisions itself, deciding whether to use the received trajectories or its locally calculated trajectories is non-trivial. This is a general problem with reaching agreement in asynchronous contexts [18]. For now, in our implementation, locally calculated trajectories are used with higher priority. A more detailed investigation to this problem is left for future work.

Overall, the post-collision trajectories agreement improves consistency of the objects after collisions, which reduces further divergence of the object states and improves the visual results of collisions.

4.4 Multi-object Collisions

The above discussions and protocols all deal with single master-replica collisions. In the motion-lock protocol, by locking the motion of the colliding objects, the two participating stations can send out collision counts before the objects collide. However, when there are multiple objects in the scene, a master and a replica that are committed to a collision can be interrupted by another replica. The second replica may suddenly change its trajectory and collide with the master. The master will be knocked off-course and the previous committed collision is no longer valid. More importantly, the potential collision counter that was sent earlier for the first collision will cause the receiving station to trigger a false collision event. The protocol is no longer correct, and the stations are then inconsistent. Figure 4.10 shows replica R_k changing its trajectory and colliding with master M_i. The previous committed collision between M_i and R_j is no longer valid.

A simple solution to the above problem is to ignore all new collisions with a master that has already locked its motion and is committed to a collision. Unfortunately, this may cause the replica that has been ignored to penetrate and pass through the master. Visually, this may confuse the player. Using the above example, if the collision between M_i and R_k is ignored then R_j will collide with M_i while R_k passes through M_i.

Instead, as a means of keeping motion-lock protocol correct and improving the visual result, we propose the Spatial-temporal bucket synchronization algorithm. Deriving from the bucket synchronization algorithm [9] where time is discretized into buckets and events that belong to the same bucket are evaluated together (see section 2.2 for details), the spatial-temporal bucket synchronization algorithm discretizes both space and time so that objects near the same collision point and time are processed together. In the next section, we present the algorithm in detail.
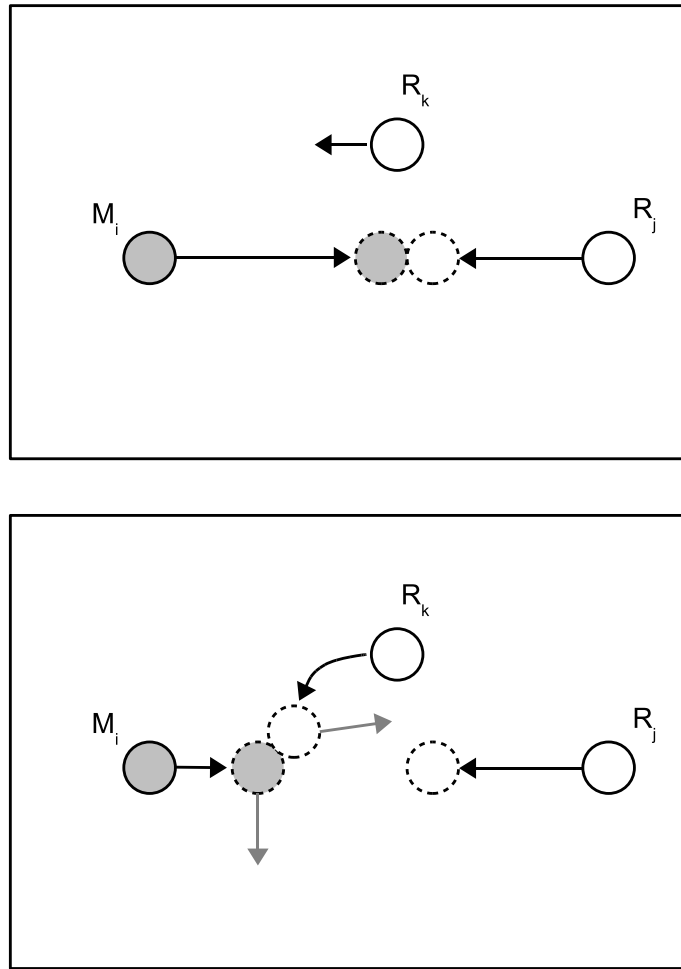
Figure 4.10: R_k interrupts the committed collision between M_i and R_j.

4.4.1 Spatial-temporal Bucket Synchronization

The spatial-temporal bucket synchronization maintains collision count consistency and improves the visual result of multi-object collisions. The committed collisions will not be interrupted so that the collision counter can still be sent early. Furthermore, replicas that are subsequently colliding with the master form a collision group with the master. The group resolves the collision together at the scheduled collision time.

The algorithm starts when a potential collision between an unlocked master and an unlocked replica is detected. The motion of the master and the replica are locked

and committed to the collision, and the potential collision count is sent to inform the other participating station about the collision. The scheduled collision time for the first master-replica collision determines the time bucket for the synchronization.

Next, any replicas that will collide with the master before the time of the first committed collision are added to the collision list and lock their motions. Instead of colliding with the master upon impact, the replicas are scheduled to resolve the collision at the end of the time bucket. The replicas in the list create a cluster of collision points around the master, and the cluster defines the spatial bucket. The scheduled collision time and potential collision counters are sent to each replica's master stations accordingly to inform about the group collision.

Finally, when the time of the first collision is reached, the collision trajectories of the master and the list of replicas are calculated and resolved together using the local states. With the post-collision trajectory agreement, if any post-collision trajectories are received, the received trajectories are used on the corresponding replicas to improve consistency and reduce correction jumps. After the collision, the master's station sends collision counters to the stations of the replicas to maintain collision count consistency.

Formally, in addition to the assumptions we made in the above protocols, we further assume the following. Let:

- $IsLocked(obj)$ return true if obj is locked.

- $collisionList$ be the list of replicas that will collide with the master.

- t_{bucket} be the collision time for the master and the listed replicas.

Algorithm 4 shows the pseudocode for the Motion-Lock protocol with Spatial-temporal Bucket Synchronization.

Evaluation of Spatial-temporal Bucket Synchronization

Replicas that join the collision group do not have the same collision point and time as the first master-replica collision. At the end of the bucket interval, these replicas

Algorithm 4 Motion-Lock protocol with Spatial-temporal Bucket Synchronization

On initialization at A

    for each replica R_j on A **do**

        $c_{i,j}^A \leftarrow 0$

        $cp_{i,j}^A \leftarrow 0$

        $t_{i,j}^A \leftarrow \infty$

        $ts_{i,j}^A \leftarrow 0$

        $c_{i,j}^B \leftarrow 0$

        $cp_{i,j}^B \leftarrow 0$

        $t_{i,j}^B \leftarrow \infty$

    end for

    $collisionList \leftarrow [\,]$ $\triangleright$ Empty list.

    $t_{bucket} \leftarrow \infty$

On each game loop at A

    for each replica R_j on A **do**

        $\Delta t_{remain} \leftarrow CollisionPrediction(M_i, R_j)$

        if $0 \leq \Delta t_{remain} \leq T$ **then**

            $t_{i,j}^A \leftarrow \Delta t_{remain} + t_{current}$

            if $IsLocked(M_i) = false$ **and** $IsLocked(R_j) = false$ **then**

                                $\triangleright$ First master-replica collision.

                $LockMotion(M_i)$

                $LockMotion(R_j)$

                $t_{bucket} \leftarrow t_{i,j}^A$ $\triangleright$ Init temporal bucket.

                add R_j to $collisionList$

                $cp_{i,j}^A \leftarrow cp_{i,j}^A + 1$

                $ts_{i,j}^A \leftarrow t_{current}$

                send $< cp_{i,j}^A, t_{bucket} >$ to B

else if $IsLocked(M_i) = true$ **and** $IsLocked(R_j) = false$ **then**

 if $t_{i,j}^A \leq t_{bucket}$ **then**

      ▷ Add subsequent colliding replicas to the list.

  $LockMotion(R_j)$

  add R_j to $collisionList$

  $cp_{i,j}^A \leftarrow cp_{i,j}^A + 1$

  $ts_{i,j}^A \leftarrow t_{current}$

  send $< cp_{i,j}^A, t_{bucket} >$ to B

 end if

end if

end if

if $t_{bucket} \leq t_{current}$ **then**

 if R_j in $collisionList$ **then**

  $CollisionResolution(M_i, R_j)$

  $c_{i,j}^A \leftarrow c_{i,j}^A + 1$

  $t_{i,j}^A \leftarrow t_{current}$

  $ts_{i,j}^A \leftarrow t_{current}$

  $t_{bucket} \leftarrow \infty$

  remove R_j from $collisionList$

  $UnlockMotion(R_j)$

  if $collisionList$ is empty **then**

   $UnlockMotion(M_i)$

  end if

  send $< c_{i,j}^A, t_{i,j}^A >$ to B

 end if

else if $cp_{i,j}^B > cp_{i,j}^A$ **and** $t_{i,j}^B \leq t_{current}$ **then**

 if $t_{current} - t_{i,j}^B < T_{cut}$ **then**

  $DelayedCollisionResolution(M_i, R_j)$

 end if

 $cp_{i,j}^A \leftarrow cp_{i,j}^B$

 $c_{i,j}^A \leftarrow cp_{i,j}^B$

$$t^A_{i,j} \leftarrow t_{current}$$

$$t^B_{i,j} \leftarrow \infty$$

$$UnlockMotion(M_i)$$

$$UnlockMotion(R_j)$$

 else if $c^B_{i,j} > c^A_{i,j}$ **then**

  if $t_{current} - t^B_{i,j} < T_{cut}$ **then**

   $DelayedCollisionResolution(M_i, R_j)$

  end if

$$c^A_{i,j} \leftarrow c^B_{i,j}$$

$$cp^A_{i,j} \leftarrow c^B_{i,j}$$

$$t^A_{i,j} \leftarrow t_{current}$$

$$UnlockMotion(M_i)$$

$$UnlockMotion(R_j)$$

 end if

 if $t_{current} - ts^A_{i,j} > T_{beat}$ **then**

$$ts^A_{i,j} \leftarrow t_{current}$$

  send $< c^A_{i,j}, t^A_{i,j} >$ to B

 end if

end for

On receipt of a $< cp^B_{i,j}, t^B_{i,j} >$ *from* B

 store $cp^B_{i,j}$ and $t^B_{i,j}$ at A

On receipt of $< c^B_{i,j}, t^B_{i,j} >$ *from* B

 store $c^B_{i,j}$ and $t^B_{i,j}$ at A

do not bounce off at their collision point since the replicas need to travel a bit further until the collision time. This may cause the replicas to slightly overlap with the master. When there are many objects colliding together, the replicas may appear to overlap and group around the master.

Fortunately, since objects are locked when the collision time is within 0 and T_{lock}, and since T_{lock} is short, the collision points and times of each master-replica collisions are spatially and temporally close to the first collision. Therefore, the replicas would not move far before the collision, and thus the overlapping is small. Furthermore, humans do not perceive collision points accurately [22]. The small difference in collision point and overlapping may not be observable. Thus, by allowing the objects to resolve the collision at the same time should not produce perceivable visual confusion.

On each station, the collision point of the group is centralized around the master. Because the masters of the stations are located at different positions, the grouped collision points among the stations will be different. Different stations may also resolve multiple master-replica collisions in different orders. The spatial and temporal bucket may thus be different among the stations, and the synchronization may potentially increase the deviation error. More detailed description on multi-object collision are discussed in section 5.4.

4.5   Conclusion

The Post-Collision protocol and the Motion-Lock protocol improve collision count consistency for distributed architectures. In terms of bandwidth, both protocols require a minimal amount of additional data to be sent between the participating stations. Collision counts are only required to be sent when a collision is detected and on each heart-beat interval. Furthermore, while the centralized server in the client/server architecture can be overloaded by a large amount of collision messages, the Post-Collision and Motion-Lock protocols distribute the processing and bandwidth usage of each collision to the relevant collision participating stations.

The Pre-collision protocol demonstrates the infeasibility of reaching an agreement

within a preset amount of time in an asynchronous network. Although the protocol can achieve a zero collision inconsistency interval, it does not maintain Δt_{col}-collision count consistency.

The Motion-Lock protocol improves upon the post-collision protocol by locking object motions and sending collision counts early so that the stations can agree on collision counts with inconsistency intervals less than the network delay. Furthermore, the Motion-Lock protocol allows post-collision trajectories to be sent early. Stations that missed the collisions are expected to benefit from the received post-collision trajectory. Overall, the motion-lock protocol should improve consistency in collision detection and resolution in distributed architectures.

The spatial-temporal bucket synchronization allows motion-lock protocol to continue to be functional in the presence of multi-object collisions. Although the manipulation of the collision point and time can increase the deviation error, the synchronization removes the visual object penetration caused by collisions between locked and unlocked objects.

Chapter 5
Simulations and Analysis

In this chapter, we analyse the effectiveness of the Motion-Lock protocol in distributed collision detection and resolution. We implemented an offline and an online simulator to analyse the protocol. The offline simulator contains a simulated network that can change the latency and packet loss rate, which allows us to test the protocol in different network condition. The online simulator builds on top of Quazal's NetZ multiplayer online middle-ware, a practical context well-known in the gaming industry. The online simulator allows us to test the protocol under real network conditions and to measure the actual bandwidth used in the protocol.

Both the offline and online simulators use Position History-Based Dead Reckoning (PHBDR) protocol to synchronize the motion states of the objects, and we build our Motion-Lock protocol on top of the PHBDR protocol. In our analysis we then compare the control protocol, which only contains the PHBDR, with the Motion-Lock protocol, which has both protocols. In addition, we implemented the Post-Collision protocol to show the improvement of the Motion-Lock protocol over this protocol. The Post-Collision protocol is implemented in both offline and online simulator. For the offline simulator, we also implemented a simple Super-node protocol such that a station is assigned to be the authority over all collision outcomes. The Super-node protocol represents a more centralized approach to contrast the distributed approach of the Motion-Lock protocol.

To analyze how the objects behave in each protocol, we set up scenarios with different object movements and, for the offline simulator, with different network conditions. The protocols are then evaluated and compared qualitatively and quantitatively. In the qualitative analysis, we observe how objects in each protocol behave and record any perceivable visual artefacts. To augment these subjective assessments we also quantitatively measure three properties of each protocol: collision counts, collision inconsistency intervals, and post-collision deviations. The post-collision deviation is calculated by summing deviations of the replica within a given interval.

In the following sections we first present the two simulators and how the protocols are implemented in the simulators. Next we discuss how the three different quantitative measurements are recorded and the significance of the measurements. Third, we discuss the scenarios and the expected object behaviours in each scenario. Finally, we present the experiment results.

5.1   Offline Simulator Design and Implementation

We built the offline simulator with adjustable network latency and packet-loss rate in order to test the protocols under different network conditions. In this section we first present the overall design and specification. Next we discuss each component in detail, and finally we show how the protocols are implemented.

The simulator uses the Discrete Event System Specification (DEVS) formalism [33] to model the system. DEVS is a hierarchical formalism for modeling systems. A system defined by DEVS is composed of subsystems. Each subsystem can be an *atomic* DEVS or a *coupled* DEVS, and the subsystems communicate with each other by sending and receiving output and input events. Each atomic DEVS is a state machine such that the state changes are triggered by internal and external transitions. The internal transitions are scheduled with a time advance. When the time expires, the state is changed and an output event is sent to other subsystems. When an input event is received, the external transition is triggered to change the state. A coupled DEVS is a subsystem that can be composed of other coupled or

64

atomic DEVSs, and simply passes the inputs and outputs between its subcomponents and other connected DEVSs. It provides a more modular structure to the system.

DEVS was chosen to model the offline simulator because it provides modular structure similar to a real distributed system, where each station can be seen as a subsystem. Furthermore, DEVS subsystems communicate with each other by sending output events, which is similar to stations sending asynchronous packets.

Our design consists of five subsystems: the System, Network, Monitor, and two Stations. The System is the top-most coupled DEVS that processes the command line arguments and starts up the other coupled DEVS. The Network coupled DEVS controls the latency and packet-loss rate for each connection between the two stations. The Station coupled DEVS initiates and updates the game loop. The Monitor is an atomic DEVS that collects statistical data from the Network and the Stations. Figure 5.1 shows the overall design of the offline simulator. The simulator is implemented in python using the Python DEVS package [3].
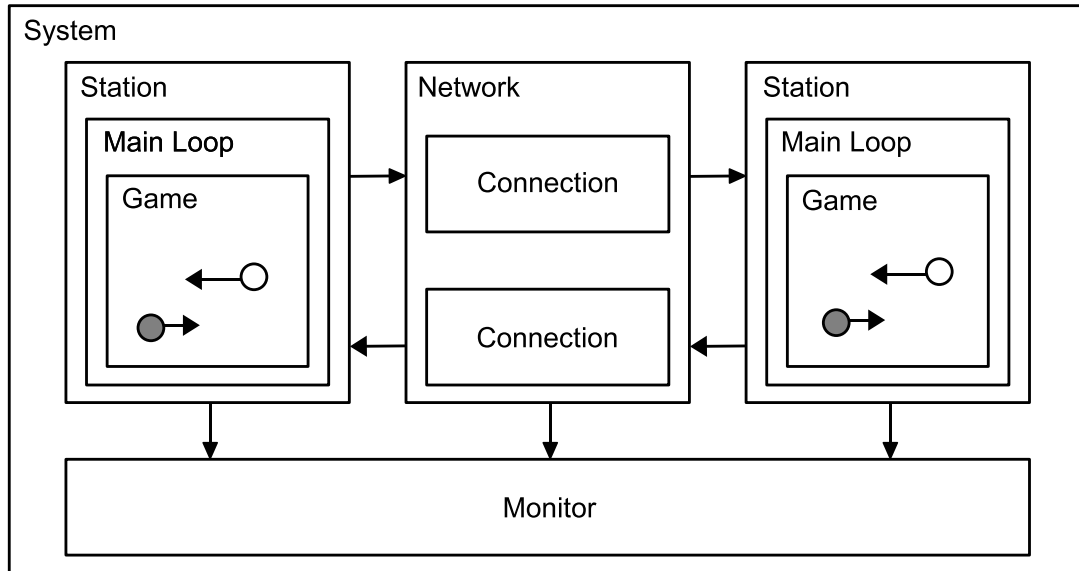

Figure 5.1: Offline simulator overall system design.

5.1.1 Station

Each Station coupled DEVS contains a Main-Loop atomic DEVS that drives the game in an update/sleep cycle. The Main-Loop atomic DEVS contains the Game state. After each update, the Main-Loop sends packets through the network to the other station. At any point of the cycle, the Main-Loop can receive packets from the network and stores them locally in a buffer.

The StationCDEVS, MainLoopADEVS, and Game classes implement each their respective components. StationCDEVS is responsible for forwarding packets between the MainLoopADEVS and the Network. It also forwards statistical data to the Monitor.

The MainLoopADEVS is the heart of the system that runs the Game model and sends updates to the other station through the StationCDEVS. The Main-LoopADEVS has five states: START, UPDATE, SENDING, and SENDONE. After initialization, the model changes from START to UPDATE with zero time advance and triggers the Game model to update the game objects. The time advance of the UPDATE state is equal to the update time. The update time determines the delta time of the Game model. The Game model moves the game objects in accordance with the delta time elapsed. The smaller the delta time, the smoother the game object moves. When the update time expires, the MainLoopADEVS changes from the UPDATE state to the SENDING state. The output function of the transition from UPDATE to SENDING renders the game objects in the graphic interface. The MainLoopADEVS then cycles between SENDING state and SENDONE state. Each transition from SENDING to SENDONE outputs one packet to the network. The cycle continues until all queued packets are sent. Note that during transitions of the internal states the MainLoopADEVS can receive packets from the network, which then triggers the external transition function. The external transition function stores the data from the packets into a buffer. Upon each UPDATE, the Game processes all the data in the buffer. Figure 5.2 shows the design of the StationCDEVS and MainLoopADEVS.

The MainLoopADEVS consists of the Game, a simple 2D physics simulation that
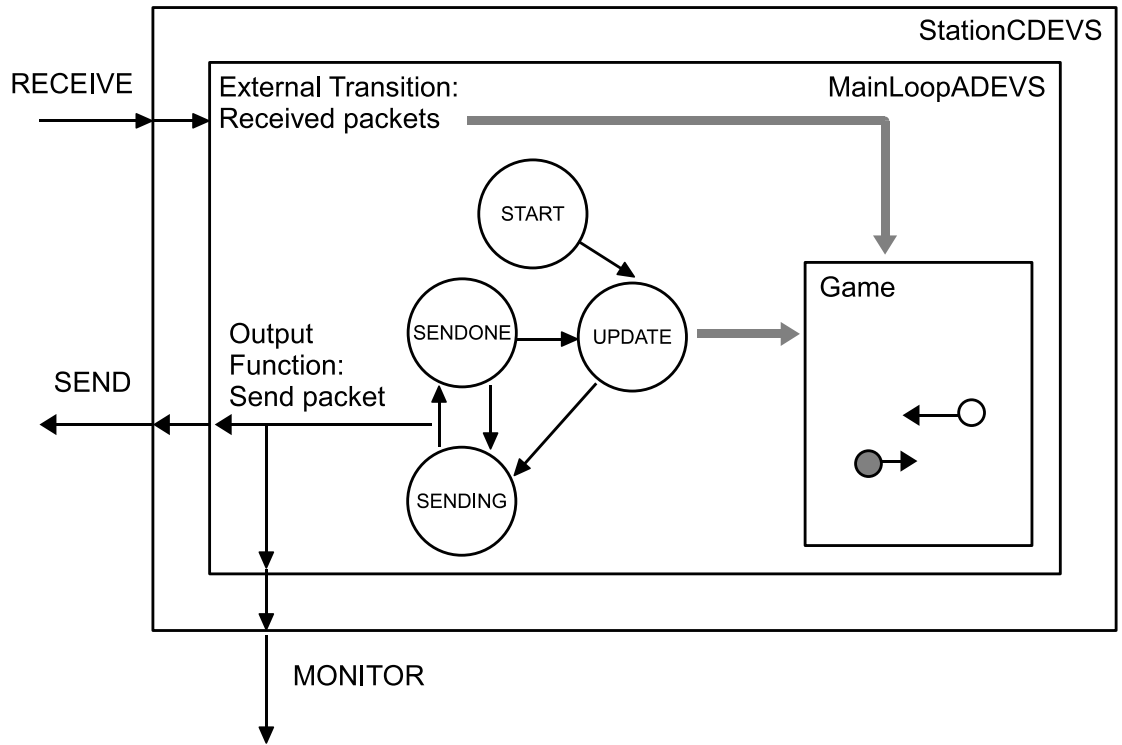
Figure 5.2: Design of StatioCDEVS and MainLoopADEVS.

contains circular objects moving and colliding with each other. The Game is modeled with a continuous spatial distribution, such that the objects are moving through continuous time and state respecting the equations of motion. However, since continuous time is difficult to model, the Game abstracts the time by discretizing it. Continuous time is broken into discrete segments of the same size as the frame interval; the smaller the frame interval the smoother the rendering of the game. With discretized time the Game can then be connected to the MainLoopADEVS.

The Game itself consists of four major components: the motion-prediction engine, the physics engine, the graphics engine, and the game object(s). The motion-prediction engine provides an interface for the motion-prediction protocols. Our current simulator has the DIS version of the dead reckoning protocol and the PHBDR protocol implemented. The motion-prediction protocols are modular units interacting through a well-defined interface; the motion-prediction engine can then switch

between different protocols with ease. The physics engine is responsible for calculating the motion of the objects, collision detection, collision resolution, and for providing a list of pre-calculated object trajectories to automate the objects' motion. The distributed collision detection and resolution protocols are also implemented in the physics engine. The graphics engine simply renders the game; our current implementation uses the TKinter package to render the game.

Each game object is composed of three bodies: the net body, physics body, and the scene body. For the masters, on each update, the physics engine takes the states of the physics bodies to calculate new states and check for collisions. The states of the physics bodies are then used to update net bodies and scene bodies. The net bodies are processed by the motion-prediction engine and sent to the network. The scene bodies are processed by the graphics engine to render the objects. For the replicas, the motion-prediction engine uses the current set of received states to calculate and update the net bodies. The physics engine updates the physics bodies. Then, depending on the current game progression, either the net body or the physics body is used to update the scene body. For example, while under normal circumstances the net body provides the motion-predicted state, if a collision is detected, the physics body will be used to update the scene body instead.

5.1.2 Network

The Network coupled DEVS consists of two Connection atomic DEVSs. Each Connection atomic DEVS receives packets from the sending station and assigns a latency value to each packet. Once the latency time is reached, a packet-loss filter determines whether the packet will be forwarded to the receiving station. Both latency and packet-loss rate can be adjusted to alter the simulated network condition.

The NetworkCDEVS and ConnectionADEVS classes implement the network and connection components respectively; Figure 5.3 shows the design. The NetworkCDEVS is connected to the two Stations and the Monitor. Packets received from Stations are forwarded to the ConnectionADEVS, and packets received from the Connection-ADEVS are forwarded to the Station to update the replicas and to the Monitor to

68

calculate statistical data.
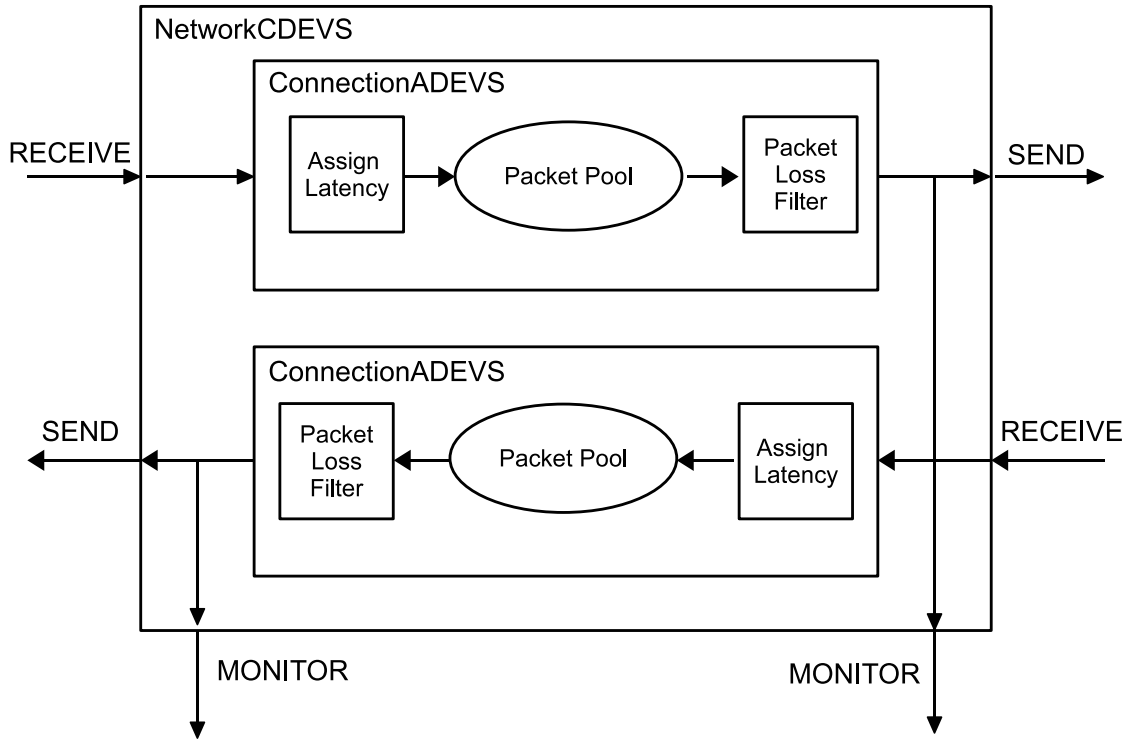
Figure 5.3: Design of NetworkCDEVS and ConnectionADEVS.

The class ConnectionADEVS receives and stores packets in a pool, assigning each packet a latency value; this value is determined using a uniform random number generator with a preset range. In early experiments, Poisson and normal distributions have been tested but yield no significant differences. The packet with the smallest latency value determines the next time advance for the model. When the packet with the smallest latency value is ready to exit the pool, a packet-loss filter with preset loss percentage determines whether the packet is forwarded to the receiving station or not. Both success and lost packet are sent to the Monitor for statistical analysis.

5.1.3 Monitor

The Monitor is atomic DEVS and implemented in class MonitorADEVS. The Monitor collects and records the game states from both Stations, and network data from

Network. From the Stations the replica deviation, collision count and collision inconsistency interval at each frame are recorded. From the Network the packet latency and loss rate are recorded. For each measurement statistical data, such as average and standard deviation, are calculated. All data are stored in text files for graph generation.

5.2   Online Simulation Design and Implementation

To prove the feasibility of the Motion-Lock protocol in a real online multiplayer game context, we implemented the protocol in the NetZ online multiplayer middle-ware. NetZ is a distributed state management framework written in C++ for online games. It provides a high-level, easy-to-use API abstracting the networking details from the game developers. The framework includes object replication, object data definitions, state synchronization techniques, fault-tolerance algorithms, and communication protocols [26]. In the previous chapters we have already discussed the different state-synchronization techniques and consistency protocols. In this section, we focus our discussion on object replication and the data definition, and how we implemented our protocol with respect to these two concepts.

The NetZ framework package also provides sample programs to demonstrate its functionality and flexibility. One of the samples program is a 3D physics simulation that contains a rectangular, enclosed arena with spheres moving and colliding. The spheres' movements can be controlled by players or automated with a preset pattern. The state synchronization techniques provided in NetZ can be turned on or off inside this sample program. We used this sample program as the base for our online simulator and built our protocols on top of it.

5.2.1   Object Replication

Many online games use a client/server architecture such that the server has authority over game state. In terms of object states, the clients send commands and events to the server, and then the server calculates new states and sends them back to the

70

clients. In the virtual simulation community, distributed virtual simulations replicate objects at each station, and allow each station to optimistically calculate and update states locally. To synchronize the states, station broadcasts the object states to update the remote replicas.

NetZ uses the replication techniques in distributed virtual simulation and applies them to multiplayer online games. NetZ's Duplication Space technology [25] uses a publish-subscribe model to automate the replication process. First, we defined the duplication space as a grouping that determines which station gets a replica. Objects are assigned as publisher, subscriber, or both. The subscribers subscribe their local stations to a duplication space, and the publishers replicate themselves at the subscribed stations. In other word, publishers are replicated only at subscribed station within the same duplication space.

One of the key concerns when we design the Motion-Lock protocol is to minimize the use of bandwidth. When a collision is detected, data should only be sent to the other collision participating station, and need not be broadcast to the other stations. In the *Data Description Language* (see next section), we define a new duplication space: Collision Space. Upon initialization, stations create a Collision Space for each master-replica object pair, and the two collision participating stations corresponding to the object pair are subscribed and published to the Collision Space. Subscriptions to the space do not change over time. For example, if there are one master and three replicas on a station, the station would be belonged to three different Collision Spaces and send the data accordingly. By creating Collision Spaces, observing stations will not receive data for collisions that the stations have no authority over.

5.2.2 Network Data Definition

In the NetZ framework, game data that will be sent through the network are defined using the *Data Description Language* (DDL) [25]. The Data Description Language uses a C++-like syntax to define the type and name of the data and group the data into datasets. Policies such as reliable/unreliable transmission and update frequency can also be defined for each dataset. Datasets are grouped into a duplication class.

71

The duplication classes are then compiled by the DDL compiler to generate optimized code for marshalling/unmarshalling and sending/receiving the data.

The provided 3D physics simulation uses a PHBDR protocol to update the replicas. Since the PHBDR protocol only sends master positions, only the positions are required to be defined in the DDL. Objects positions are defined as the Extrapolated Position dataset, which contains three floating point values for the 3 dimensional Cartesian coordinates. The dataset uses unreliable UDP protocol for transmission, and uses the PHBDR protocol's default error threshold to control the update frequency. Lastly, the Net Object duplication class contains the Extrapolated Position and other datasets that are required to create and update the replicas.

The Motion-Lock protocol requires also sending the collision counters, potential collision counters, collision times, and post-collision trajectories. In DDL, we defined the Collision Data dataset that contains integers for the collision and potential collision counters, floats for the collision time and the post-collision trajectory vectors. The Collision Data is sent with unreliable UDP for fast transmission. The update frequency depends on when a collision is detected and the preset heart-beat frequency. The frequency is controlled by the Motion-Lock protocol, so we do not define it at the DDL level. We also define a Collision Index dataset that contains the two colliding object's IDs associated with the Collision Data. The IDs do not change, so the update policy is constant and only sent once at initialization. The two datasets are grouped into the Collision Object duplication class. The stations then create a Collision Object for each object pair.

5.3   Offline Experimental Analysis

The offline simulator was build to allow us to analyze the protocols with more control over many different aspects. With the adjustable, simulated network, we can observe how the protocols behave in, not just normal conditions, but also extreme network conditions. With the monitor atomic DEVS, data generated from the stations can be easily recorded and analyzed. Furthermore, frame rates for the stations can be

synchronized to allow the object states to be sampled at the same time. This increases the accuracy of the calculated statistical data.

For the offline simulation, we implemented the control, Super-node, and Post-Collision protocols to compare with the Motion-Lock protocol. The Motion-Lock protocol includes post-collision trajectory agreement. Spatial-temporal bucket synchronization is not implemented because the offline simulator is only testing single collisions.

- The control only has the underlying PHBDR protocol to synchronize the motion states. No collision agreement mechanism is implemented. The stations will be collision count inconsistent if one station missed a collision. It acts as a basis for the purpose of comparison.

- The Super-node protocol assigns one station to be the authority over all collision detections. All stations still contain masters and replicas, and the motion of the master on each station, including the Super-node, is still calculated locally. Only when the objects collide, the detection and resolution of the collision is determined by the Super-node. If the Super-node detected a collision, it will send collision counts to trigger collision events in the other stations. This protocol represents a more centralized approach to maintain collision consistency. In our implementation, we assign station A to be the Super-node.

- The Post-Collision protocol sends out the collision count after a collision is detected. The collision participating stations will eventually be consistent, but the collision inconsistency interval varies with the network latency. We want to compare with the Post-Collision protocol to evaluate the effect of reducing the collision inconsistency interval in the Motion-Lock protocol.

In this section, we present the experimental setup and results for the protocols. First, the details of the test scenarios are presented. Next, the measurements are discussed. Finally, the qualitative and quantities results are presented.

73

5.3.1   Experiment Setup

We set up six scenarios to evaluate the protocols. All scenarios involve two stations, A and B, such that A contains master M_1 and replica R_2 and B contains master M_2 and replica R_1. The objects are represented by circles with radius r equals to 10 pixels:

Linear-Linear-Collide (LLC)

In this scenario, the objects are moving in straight line towards each other, and will collide head on at one point. The scenario evaluates how the protocols deal with objects in simple linear motions and head on collisions. Since linear motion follows the motion model, states of the replicas are correct. Figure 5.4 depicts the LLC scenario.
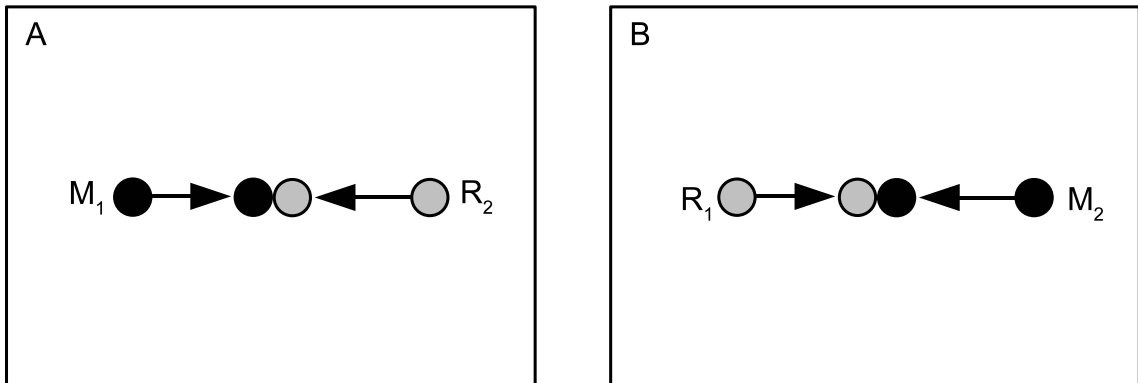

Figure 5.4: Linear-Linear-Collide scenario.

Linear-Linear-Pass (LLP)

In this scenario, the objects are moving in straight line, similar to the above scenario, except that the objects will not collide and instead will just miss each other by $3r$, from center to center. Figure 5.5 depicts the LLP scenario.

Figure 5.5: Linear-Linear-Pass scenario.

Circular-Linear-Collide (CLC)

In this scenario, M_2 is moving in straight line while M_1 is moving in a circle to simulate more complex motion, such as from player inputs. The extrapolated state of the replica R_1 of M_1 is thus inaccurate. The objects will collide at one point; however, R_1's extrapolation error may cause it to miss the collision with M_2. This scenario is to evaluate how the protocols deal with missed collisions. Figure 5.6 depicts the CLC scenario.
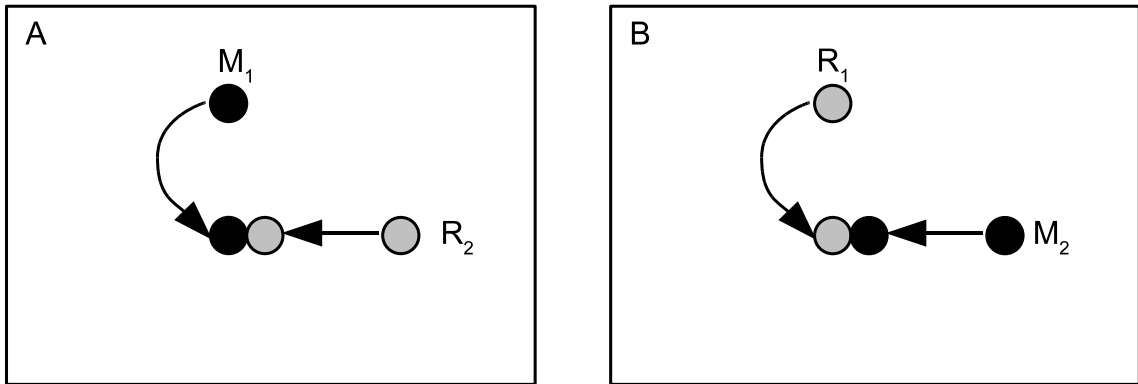
Figure 5.6: Circular-Linear-Collide scenario.

Circular-Linear-Pass (CLP)

Motion of the objects is similar to CLC except the objects will not collide, and instead will just miss each other at one point by $3r$, from center to center. Again, the state of R_1 is inaccurate and may cause false collisions. This scenario is to evaluate how the protocols deal with false collisions. Figure 5.7 depicts the CLP scenario.

Figure 5.7: Circular-Linear-Pass scenario.

Circular-Circular-Collide (CCC)

In this scenario, both objects are moving in circles and the objects will collide at one point. This is to evaluate how the protocols perform when the states of both replicas are inaccurate. Figure 5.8 depicts the CCC scenario.

Figure 5.8: Circular-Circular-Collide scenario.

76

Circular-Circular-Pass (CCP)

This scenario is similar to the CCC scenario except the objects do not collide but just miss each other by $3r$, from center to center. Figure 5.9 depicts the CCP scenario.
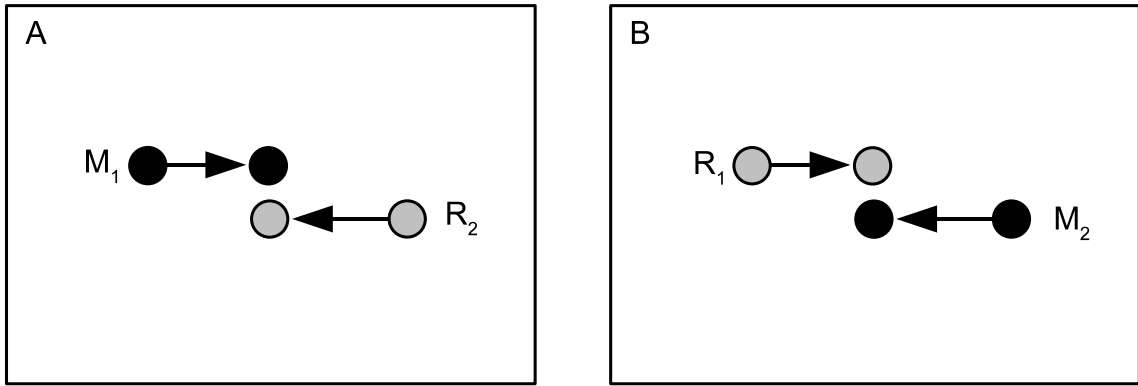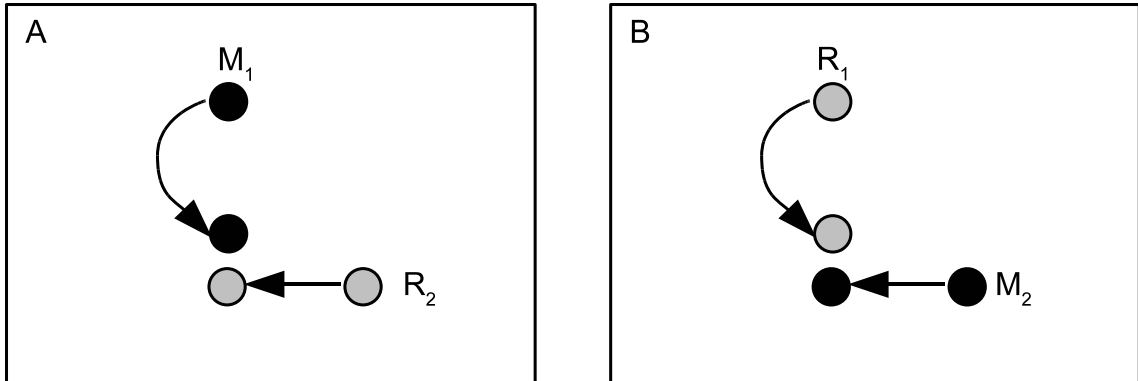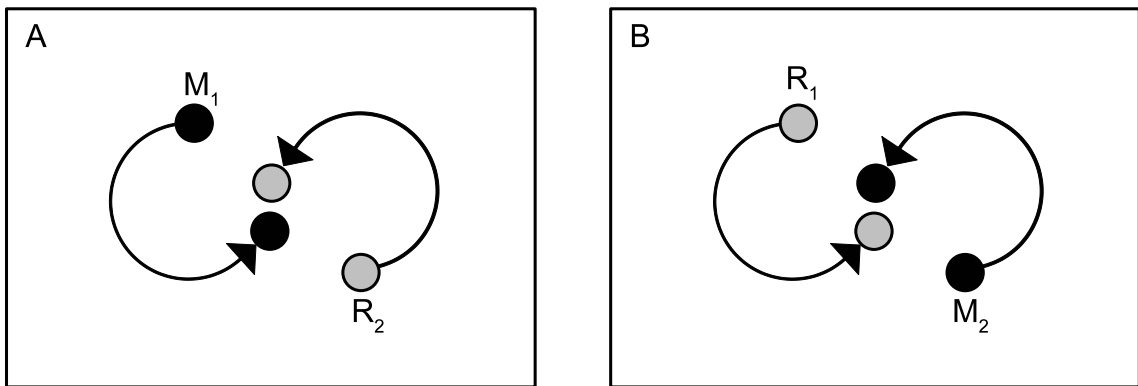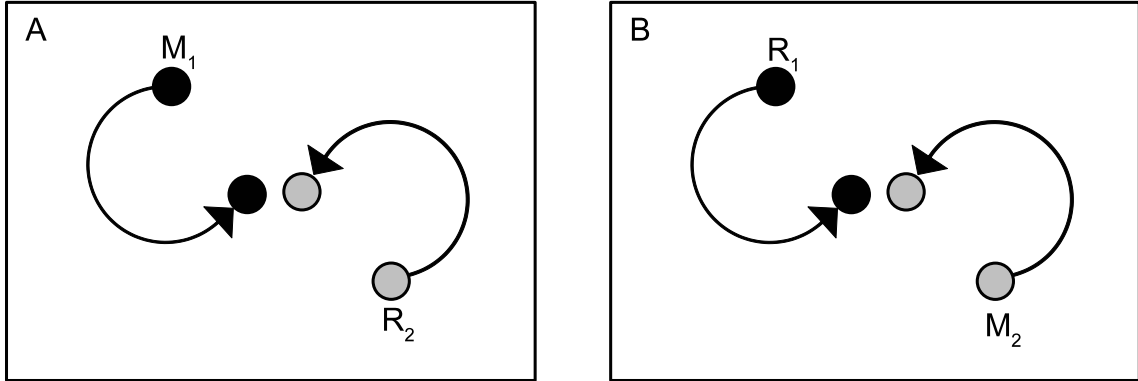

Figure 5.9: Circular-Circular-Pass scenario.

Network Conditions

Three different network conditions are setup to combine with each scenario:

- The good network condition has 50ms latency and 10% packet-loss rate.

- The moderate network condition has 100ms latency and 20% packet-loss rate.

- The congested network condition has 150ms latency and 40% packet-loss rate.

The values for the network conditions are chosen to closely match the real network. According to Pantel et al. [23], network latency beyond 200ms may cause the game to be unplayable; therefore, we only test the protocols in conditions such that the game is still playable.

Each combination of the scenarios and conditions is run 50 times with different random seeds. If a collision is detected the run continues for 500ms more before it terminates; if there is no collision the run terminates after 3 seconds. The machine used for the offline simulator is an Intel Core 2 2GHz machine with 2GB of memory and runs on 32 bit Windows Vista.

5.3.2   Measurements

For the quantitative analysis, we measured the number of collisions detected on each station to determine the collision count consistency, the time of collision on each station to determine collision inconsistency interval, and the positions of the objects after collision to determine the post-collision deviation.

Collision Count

The number of collisions detected in each station is recorded to determine if the stations are collision count consistent, and to show the importance of collision count consistency techniques. Whenever a collision is detected, the collision count is sent to the other participating station and the monitor. The monitor records the collision count for further analysis. The Motion-Lock, Super-node, and Post-Collision protocols are expected to have equal collision counts between the stations, and the Control may or may not have equal counts since no collision consistency algorithm is implemented.

Collision Inconsistency Interval

The collision inconsistency interval is measured to show that the Motion-Lock protocol can effectively minimize the interval. The monitor records the time the collision was detected or informed of through the collision counters, and then the monitor calculates the difference in collision times between the stations to determine the collision inconsistency interval. At the end of each run the sum, average, standard deviation, minimum, and maximum of the collision inconsistency intervals are calculated for statistical analysis.

The average collision inconsistency interval for the Motion-Lock protocol is expected to be lower than the Post-Collision protocol, and, more importantly, lower than the preset network latency. For the Control, the average value is expected to be large since no collision consistency is implemented.

Post-collision Trajectory Deviation

The size of the post-collision trajectory deviation indirectly determines the size of the correction jumps (or convergence times and distances). By measuring the post-collision trajectory deviation we can determine which protocol minimizes any visually confusing corrections. To calculate deviations the positions of the masters and the replicas are recorded by the Monitor. The Monitor then determines the deviation of the replica by calculating the distance between position of the master and the replica at matching time stamps. To determine the post-collision deviation, after each collision the deviations are summed from the time of collision to the end of the 500ms interval, so that the runs have the same summation interval for better comparison. At the end of each run, the sum, average, standard deviation, minimum, and maximum of the post-collision trajectory deviation are calculated for statistical analysis.

The Motion-Lock protocol is expected to have a lower post-collision deviation than the Post-Collision protocol because the trajectories are pre-calculated. On the other hand, the Super-node protocol may have better performance because of the centralized authority.

5.3.3   Qualitative Result

One of the goals of the Motion-Lock protocol is to produce visually pleasing collisions and reduce visual anomalies. We therefore also compare the protocol qualitatively by examining the behaviour of the objects in each scenario. Although this is a subjective assessment, the behaviours we remark on are clear and supported by the quantitative evidence as well.

Control

In both LLC and LLP scenarios the objects behave optimally: the objects collide cleanly in LLC and pass by each other in LLP. In CLC, CLP, CCC, and CCP, with congested network condition, both R_1 and R_2's circular motions deviate greatly and

cause misses and/or false collisions. Because there are no agreement messages exchanged, if there is a missed collision and/or false collision the master of one station may detect a collision and bounce off, with its replicas behaving the same. On the other station, however, the master that missed the collision will continues to move at its original trajectory, and so do its replicas. This causes every station to observe one object bouncing off because of the collision while the other object ignores the collision and continues the original trajectory. This is distracting, and players may be confused as to whether there is a collision or not.

Super-node Protocol

The Super-node protocol uses station A as the central authority in detecting and resolving collisions. This results in some unsurprising asymmetry in the object behaviours. In all the scenario with normal and congested network conditions, on the Super-node A, M_1 and R_2 collide with clean results. However, on B, M_2 and R_1 penetrate each other because the collisions are not resolved locally, and so the bounce occurs with some delay. The penetration occurs even in LLC despite the simple linear motion of the objects. Station B does not resolve collisions to create different post-collision trajectories than A. However, collisions in A create sudden changes in object motions and can still cause large deviation errors. Large correction jumps therefore still occur.

Post-Collision Protocol

In the LLC and LLP, objects collide or pass by each other correctly without any visual anomalies. In CLC, CLP, CCC, and CCP, on the station that missed the collision, objects pass each other at the expected collision point. Once the collision counts are received, however, the objects bounce off with a large gap greater than $2r$ between them. Under congested network condition the gap between the objects can be very noticeable. Furthermore, the delay in resolving the collision causes different post-collision trajectories between the stations, and large correction jumps with $\Delta\theta_{col}$ greater than 90 degree are observed.

Motion-Lock Protocol

In the LLC and LLP, objects collide or pass by each other correctly without any visual anomalies. In CLC, CLP, CCC, and CCP, the collision gaps between the objects are smaller than the gaps in the Post-Collision protocol. This correlates with quantitative data presented below, where we show the collision inconsistency interval is also reduced. However, in CCP, objects sometimes appear to reduce the gap by pulling toward each other; this is a result of the motion lock period sometimes exceeding the threshold beyond which it is visually apparent. By reducing the T_{lock}, the pulling can be less apparent, but this sacrifices the ability for potential collision counters to be sent early. Some large correction jumps with $\Delta\theta_{col}$ greater than zero are still observed, although the sizes are reduced. Overall, the Motion-Lock protocol produces better results than the other protocols with cleaner collisions by reducing the gaps in between the objects, and with fewer large correction jumps.

5.3.4   Quantitative Result

Human observations are of course not enough to show the performance of the protocols. In this quantitative analysis we measure the protocols in terms of the discussed quality measurements in section 5.3.2 and analyze the significance of the observed results.

Collision Count

The number of collisions detected on each station are recorded to determine if the stations are collision count consistent. Since each scenario is tested for 50 runs, for scenarios with intended collisions, the collision counts are expected to be 50. For object passing scenarios, the expected collision counts are all 0.

In the LLC and LLP scenarios collision counts for all protocols return the expected values. In CLC, CLP, CCC, and CCP, because of the circular motion, deviations from the replicas are causing false and missed collisions. Tables 5.1, 5.2, and 5.3 show the collision count for the scenarios with good, normal and congested network condition,

respectively.

The tables show that the stations in the control protocol are inconsistent in collision count in most of the scenarios because no messages are passed between the stations to inform each other of detected collisions. On the other hand, the stations with Super-node, Post-Collision, and Motion-Lock protocols are all consistent in collision count.

In the CLP scenario where objects pass each other without colliding, stations using the Super-node protocol correctly detect no collisions because the Super-node A, moves M_1 in circular motion while R_2 is moved linearly without any deviation error, allowing the objects to pass each other correctly with no false collisions. On station B, R_1 may deviate, but since B is not the Super-node collisions caused by R_1 are ignored. In CCP, however, R_2 on A moves in a circular motion with error, and so false collisions are still detected on the Super-node A.

On the other hand, in CLP and CCP, stations with the Post-Collision and Motion-Lock protocols are detecting and agreeing to many false collisions caused by the replicas. Note that A and B are collision count consistent, because of collision count passing; the collisions, however, are unintentional. Moreover, even in good network conditions, the Motion-Lock protocol detects false collisions because the objects are locked and forced to collide. This is undesirable from a correctness perspective, but may still be acceptable in terms of fairness and consistency, since stations using the Motion-Lock protocol are detecting the same number of collisions despite the different network conditions.

In CCC and CCP, the Motion-Lock protocol can detect more than 50 collisions. This may due to the inaccuracy of the potential collision prediction algorithm. After a collision objects are still close to each other, and numerical error or insufficient time for collision resolution to move objects apart can cause the collision prediction algorithm to incorrectly detect multiple collisions instead of one. The problem can be reduced with better collision prediction, although this is left for future work.

Table 5.1: Collision Count for Good Network Condition (50ms delay, 10% loss)

Scenario	CLC		CLP		CCC		CCP	
Station	A	B	A	B	A	B	A	B
Control	50	50	0	1	50	49	1	0
Super-node	50	50	0	0	50	50	1	1
Post-Collision	50	50	1	1	50	50	1	1
Motion-Lock	50	50	50	50	50	50	50	50

Table 5.2: Collision Count for Normal Network Condition (100ms delay, 20% loss)

Scenario	CLC		CLP		CCC		CCP	
Station	A	B	A	B	A	B	A	B
Control	50	50	0	50	44	43	35	37
Super-node	50	50	0	0	50	50	46	46
Post-Collision	50	50	50	50	50	50	50	50
Motion-Lock	50	50	50	50	50	50	50	50

Table 5.3: Collision Count for Congested Network Condition (150ms delay, 40% loss)

Scenario	CLC		CLP		CCC		CCP	
Station	A	B	A	B	A	B	A	B
Control	50	10	0	25	44	42	46	46
Super-node	50	50	0	0	50	50	50	50
Post-Collision	50	50	25	25	50	50	50	50
Motion-Lock	50	50	50	50	52	52	51	51

Collision Inconsistency Interval

We compare collision inconsistency intervals among the Super-node, Post-Collision, and Motion-Lock protocols. The Control cannot be used here because, without an agreement protocol, if one station misses a collision the inconsistency time grows unbounded.

Figure 5.10 shows collision inconsistency time for the LLC scenario. The color bars and the lines extended from the bars represent the average and standard deviation of the collision inconsistency interval, respectively. With objects moving in linear motion, the collision inconsistency intervals for Post-Collision and Motion-Lock protocols is zero because both stations detect and resolve the collision independently. Furthermore, because the objects are moving linearly, no false or missed collisions occur. On the other hand, the Super-node protocol shows a long collision inconsistency interval. This is because station A needs to inform B of every collision, and thus, even for simple linear motions, the Super-node protocol shows delays in resolving collisions, as described in the qualitative analysis.

Figures 5.11, 5.12, 5.13, and 5.14 show the collision inconsistency interval for CLC, CLP, CCC, and CCP scenarios. Overall, the motion-lock protocol shows shorter collision inconsistency intervals, and even with 150ms network latency the motion-lock protocol has the inconsistency interval below the network latency. This demonstrates that our motion-lock protocol can successfully minimize the collision inconsistency interval. In 5.12, the Super-node protocol shows zero inconsistency intervals because it detected no collisions (discussed in the above collision count section).

Post-Collision Trajectory Deviation

We of course also want to reduce the deviations of the replicas after collisions to prevent large jumps when replicas correct their positions in their next state updates. The post-collision trajectory deviations provide a good indication on how large the correction jump will be. In this analysis we compare all four protocols.

Figure 5.15 shows that, again even with simple linear motion, the Super-node protocol causes delays in resolving collisions, and thus creates post-collision deviation
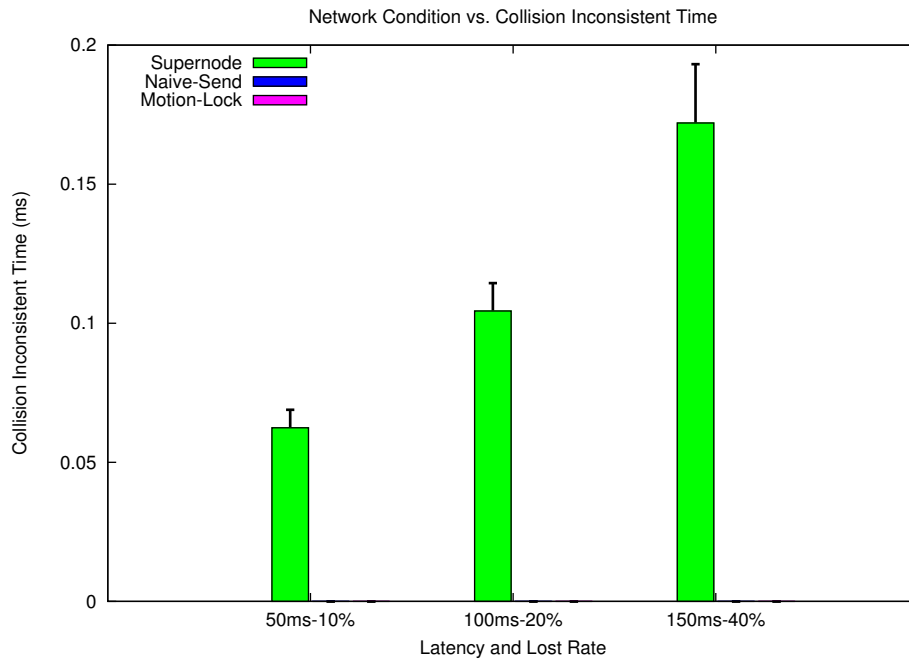
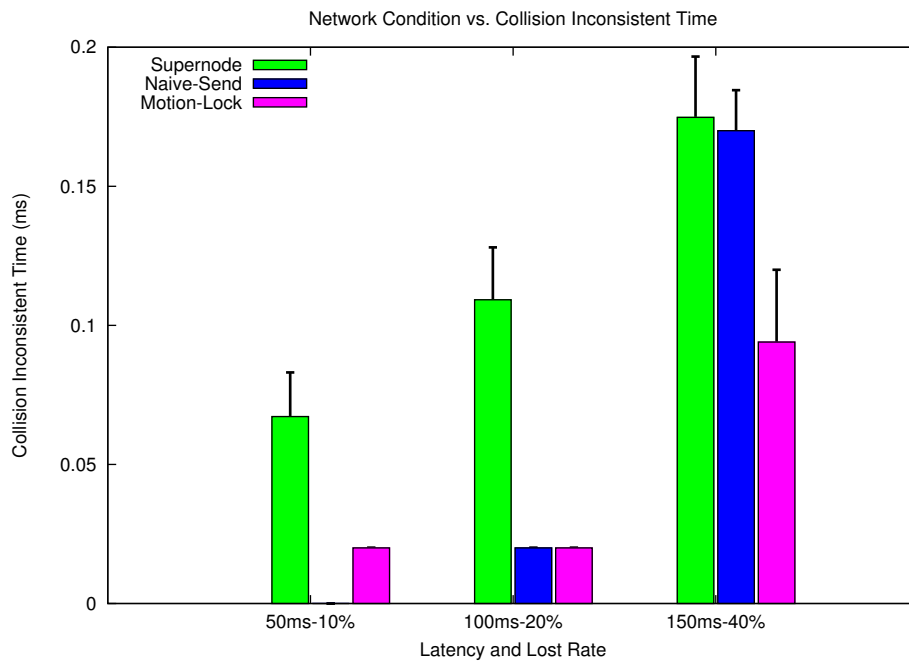Figure 5.10: Collision inconsistency interval for LLC Scenario

Figure 5.11: Collision inconsistency interval for CLC Scenario
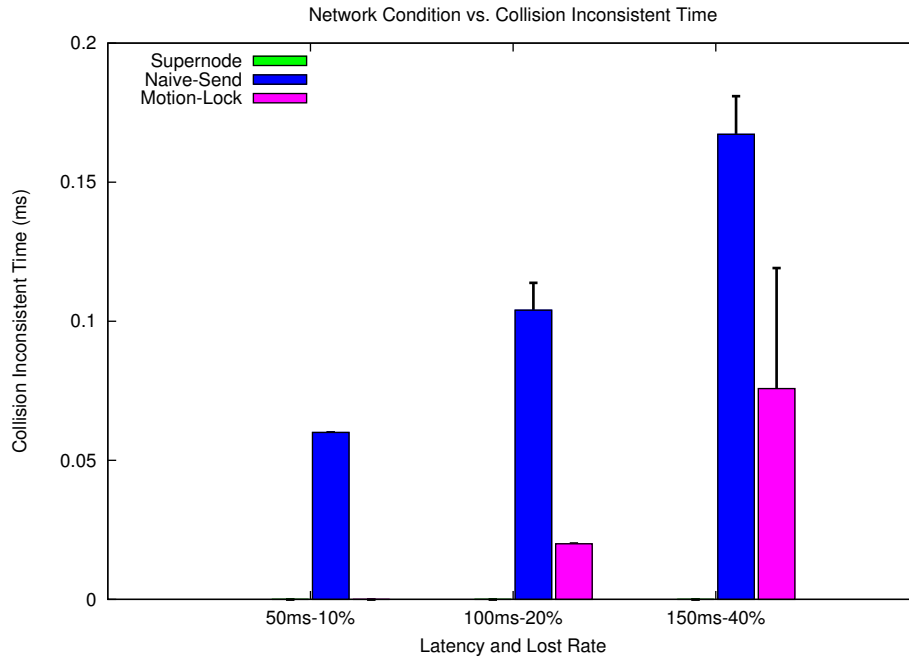
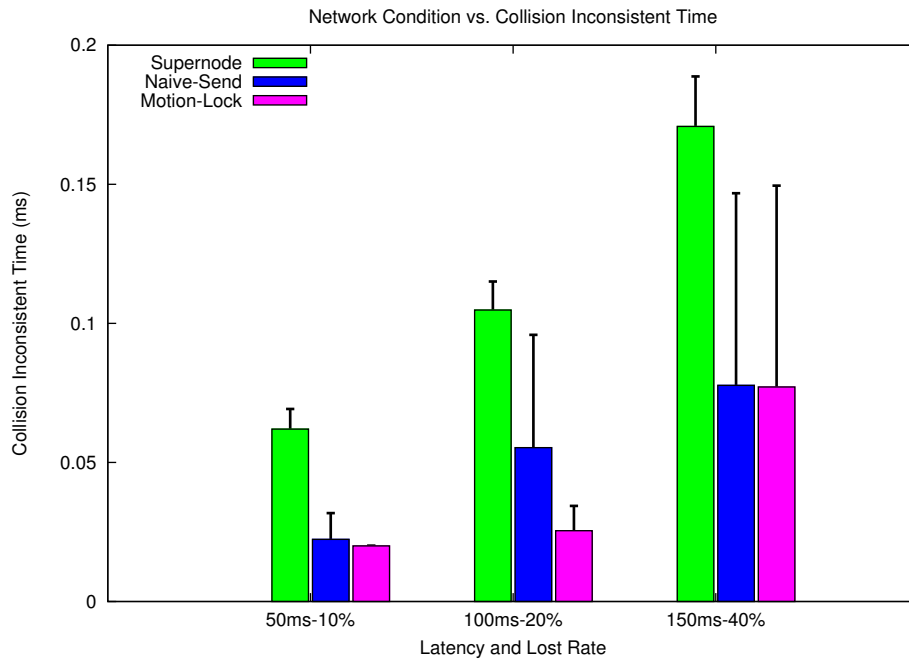Figure 5.12: Collision inconsistency interval for CLP Scenario

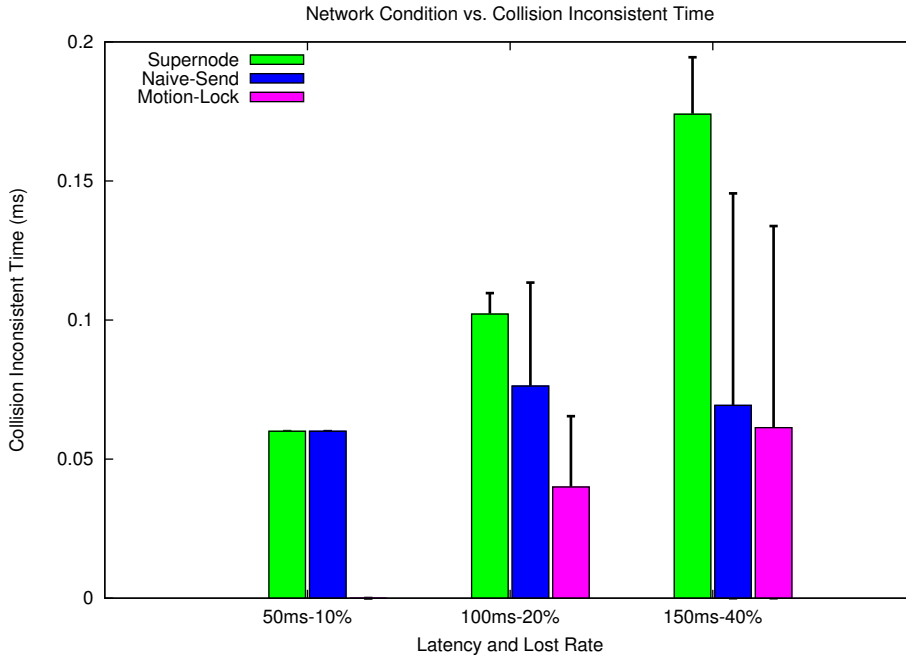Figure 5.13: Collision inconsistency interval for CCC Scenario

Figure 5.14: Collision inconsistency interval for CCP Scenario

errors for the replicas. The other protocols show no error at all under this easy to predict scenario.

Results from more complex motion tests are shown in Figures 5.16 and 5.17. Here we see that the motion-lock protocol has a smaller deviation error than all protocols, except in the case of the Super-node protocol tested on circular-linear-pass. In this situation the Super-node protocol successfully filters out the false collisions, resulting in no deviation.

Both objects moving in a circular path represents a difficult test for distributed collision detection. Figures 5.18 and 5.19 show that all protocols improve significantly over the control, with the Motion-Lock protocol having approximately the same error as others. When both objects are moving in circles, collision prediction may not accurately detect potential collisions, and so the motion-lock protocol is unable to send the collision counter prior to the actual collision time. The collision inconsistency intervals thus are about the same as the Post-Collision protocol, or the delay generated by the Super-node protocol.
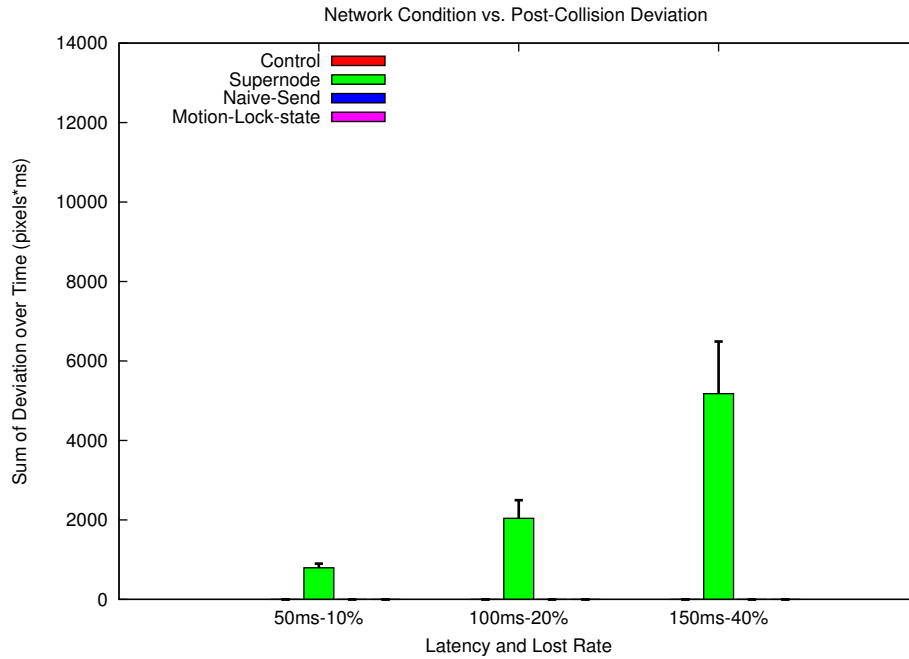
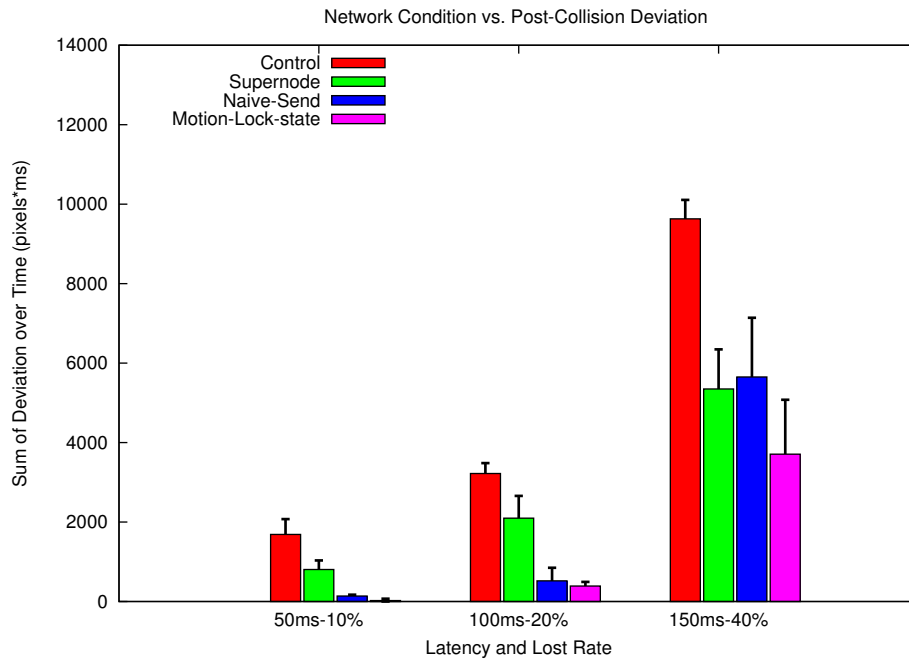Figure 5.15: Post-Collision Trajectory Deviation for LLC Scenario

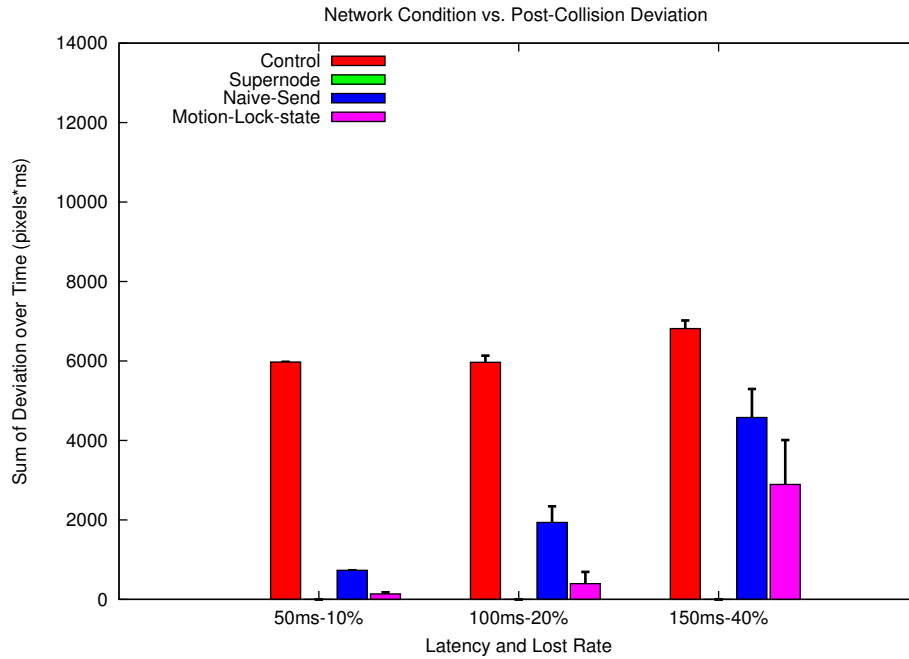Figure 5.16: Post-Collision Trajectory Deviation for CLC Scenario

88

Figure 5.17: Post-Collision Trajectory Deviation for CLP Scenario
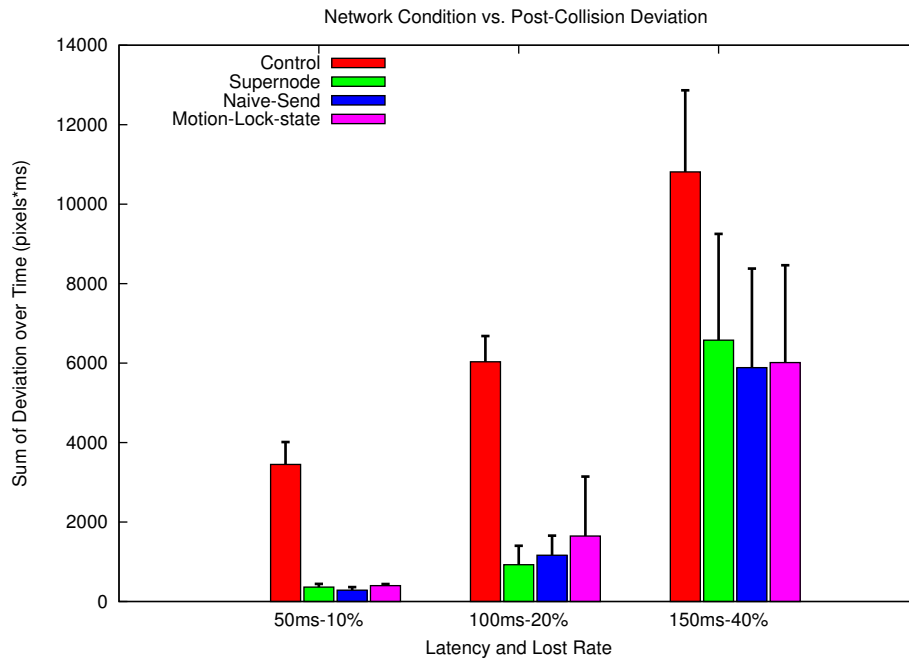
Figure 5.18: Post-Collision Trajectory Deviation for CCC Scenario
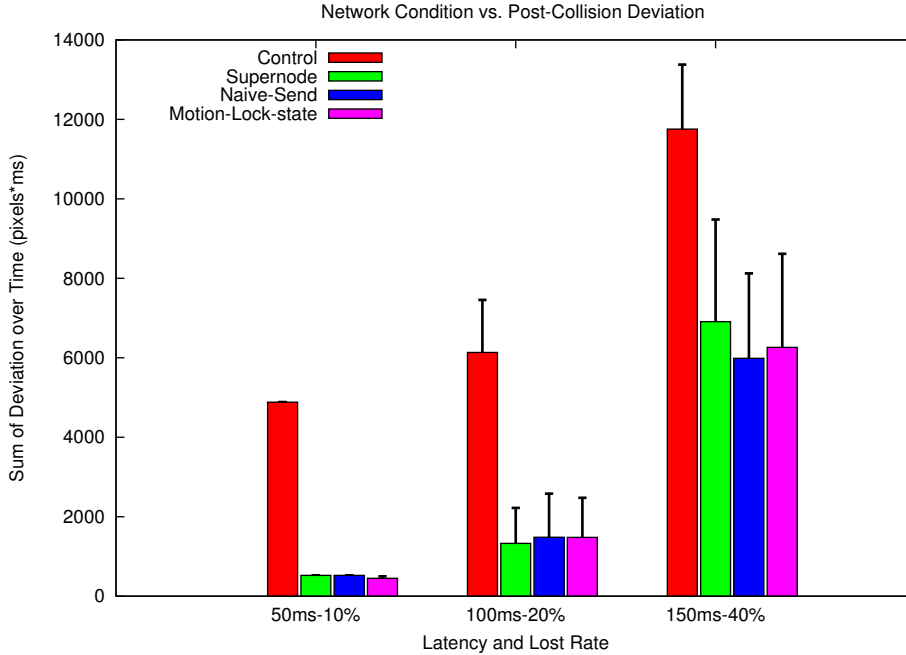
Figure 5.19: Post-Collision Trajectory Deviation for CCP Scenario

5.4 Online Experimental Analysis

In the above offline experiment we analyzed pair-wise object collision behaviour. In the online experiment we increase the complexity of the scenario by having multiple objects in the scene to create multiple collisions. This allows us to test the effectiveness of the Spatial-temporal bucket synchronization algorithm. Furthermore, with a real network context we are able to measure the amount of data sent and received at each station. This allows us to analyze the actual network impact of the protocols.

For the analysis, we focus our observation on two types of collisions: multi-object collisions, and consecutive collisions. Multi-objects collisions occur when more than two objects collide near the same point, as discussed in the Spatial-temporal bucket synchronization section 4.4.1. Consecutive collisions occur when one object collides with a series of objects consecutively at different collision points, within a short time period. Each collision contributes a certain amount to post-collision deviation, and so consecutive collisions can lead to large deviations and thus large correction jumps.

We implemented the Control, Post-Collision, and Motion-Lock protocols for the online simulation. To test the Spatial-temporal bucket synchronization algorithm we also set up two different versions of the Motion-Lock protocol, either with or without spatial-temporal bucket synchronization.

In this section, we present the experimental setup and results for the online simulation. First, the details of the test scenarios are presented. Next, the measurements are discussed. Finally, the qualitative and quantitative results are presented.

5.4.1 Experiment Setup

We set up a specific multi-object collision scenario to analyze protocol performance, and used two machines on the Internet to test actual distributed performance. One machine was located on McGill campus in Montreal, and one in Longueuil. The machine in Montreal is an Intel Core 2 2GHz machine with 2GB of memory and runs on 32 bit Windows Vista. The machine in Longueuil is an Intel Core 2 Quad 2.5GHz machine with 4GB of memory, running on 64 bit Windows 7. For simplicity, we term the machines as station A and B.

Station A contains one master while station B contains 7 masters, so each station has 8 objects to render. Initially, objects are placed far away from the center. When the scenario starts, all objects move toward the center and collide with each other to engage in a multi-object collision. Every three seconds after the main collision, user commands are simulated to force objects to once again all head for the center, repeating the multi-object collision. However, due to the nondeterministic nature of the network and extrapolation error, the replicas may not reach the center at exactly the same time. Some replicas will deviate and consecutive collisions can occur. This scenario maximizes the chance of multi-object and consecutive collisions and of observing inconsistency in the treatment of the 7 replicas on A. For each protocol, the scenario runs for 20 minutes. Figure 5.20 shows the multi-object collision where objects collide at the center. Figure 5.21 shows the consecutive collisions of the master M_1 that occur when replicas deviate and do not collide at the center.
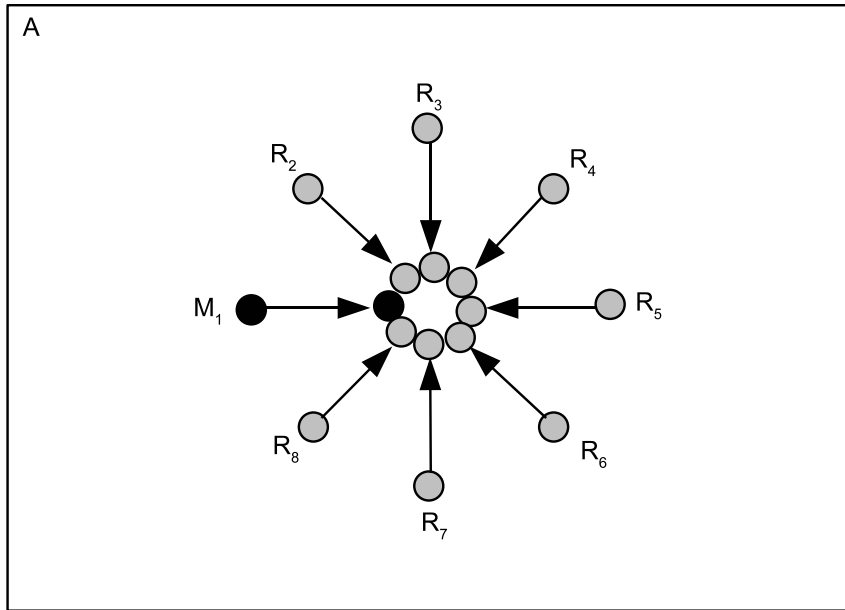
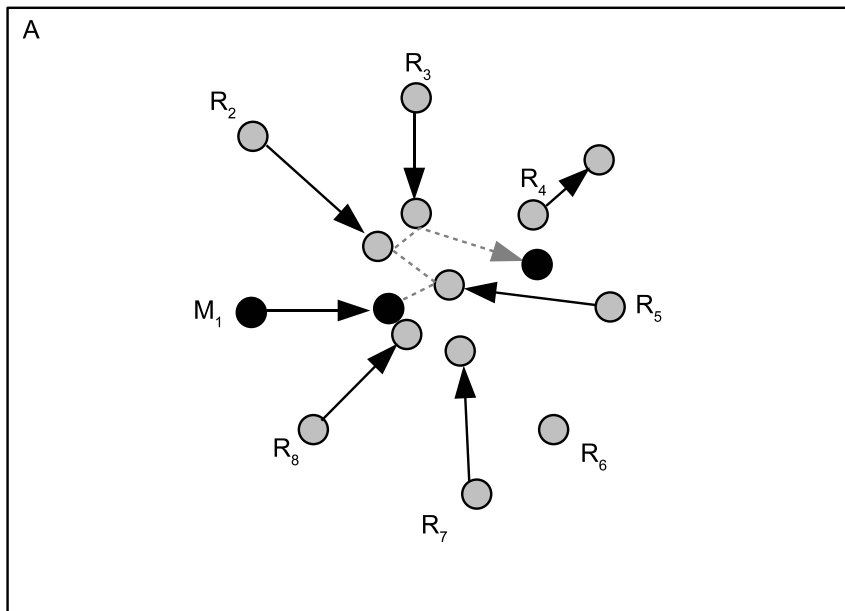Figure 5.20: Multi-object collision scenario.

Figure 5.21: Replica deviation causes consecutive collisions.

5.4.2 Measurements

Similar to the offline experiment, for the online experiment, we need to define some measurements to evaluate the protocols. The data are measured and recorded in a similar way to the offline Monitor case. However, since stations are distributed geographically, each station records the data in a local file, then the files are combined and processed after each simulation. We measure and calculate the collision count and collision inconsistency interval the same as the offline simulator. For details on these two measurements, see section 5.3.2.

Deviation

In a multiple collision scenario, each collision can be difficult to isolate in order to calculate the post-collision deviation. Instead, we sum the deviation for each object from the beginning to the end of each 20 minute run.

Deviation error is calculated from samples of object positions. Object positions are recorded every 20ms between frames. Because the stations can be running at different frame rates, the recorded positions of master and replica may not be in sync. Therefore, master positions are interpolated to match the time of their replica's recorded position. At the end of each run, the sum, average, standard deviation, minimum, and maximum of the deviation error are calculated for statistical analysis.

Bandwidth Usage

The Motion-Lock protocol is designed in a way such that the collision data sent across the network should not cause significant extra burden to the network and stations. Nevertheless, we do send extra data, and so we want to observe the amount of bandwidth used by our protocols for network efficiency analysis. The NetZ framework provides functions to measure the number of bytes sent and received at each station. We simply use these functions to record the amount of network data. The number of kBytes sent and received are sampled at every 20ms between frames for the 20min run. At the end of each run, the sum, average, standard deviation, minimum, and maximum of the bytes sent and received are calculated for statistical analysis.

Motion Locked Percentage

When there are multiple objects in the scene with many potential collisions the Motion-Lock protocol may result in the master having its motion locked very frequently; players may feel they lose responsiveness in controlling their objects. Therefore, we want to analyze the degree of control being discarded due to motion locking. We recorded the total number of automated commands that change object motion to reach the center, and the number of automated commands discarded while the objects' motion is locked. The percentages of commands that are discarded can then be calculated.

5.4.3 Qualitative Result

Similar to the offline simulation, we observe the behaviours of the objects and discuss any visual anomalies. We also focus our observation on the multi-objects collisions, and consecutive collisions.

Control

Because there is no collision count agreement, when an object encounters a consecutive collision, the master and the replicas may detect a different number of collisions. Furthermore, as each collision amplifies the deviation error, after a few consecutive collisions, the replicas can end up in a complete different location than their master. This can cause very large correction jumps as great as across the entire game arena.

Post-Collision Protocol

Similar to the offline simulation observation, we detected collision gaps greater than $2r$ between objects due to the long collision inconsistency interval. In consecutive collisions, the masters and replicas detect the same collision count, but the large collision gap can still cause replicas to deviate, and thus creating large correction jumps.

Motion-Lock Protocol

Without the Spatial-temporal bucket synchronization collisions between a locked object and an unlocked object are ignored. Therefore, in multi-object collisions we observed object penetrations. By sending the collision counter early, gaps in collisions are much smaller than seen in the Post-Collision protocol. This prevents large deviations caused by the consecutive collisions, and so the numbers of large correction jumps are also reduced.

Motion-Lock with Spatial-temporal Bucket Synchronization

With the Spatial-temporal bucket synchronization no object penetrations are observed in multi-object collisions. The spatial-temporal bucket synchronization algorithm manipulates object collision times so that they all bounce off at the same time and avoid object penetrations. However, because of the collision time manipulation, masters and replicas may end up with different post-collision trajectories. Therefore, although penetration issues improved, we observed more state correction jumps in motion-lock protocol with spatial-temporal bucket synchronization than without.

5.4.4   Quantitative Result

In the multi-object collision scenario, master M_1 on A interacts with 7 replicas from B, while the masters on B interact with only one replica, R_1 from A. Since we are interested in master-replica collisions, we focus on the experimental results involving M_1 or R_1. For bandwidth analysis, we focus on the number of bytes station A sent and received.

Table 5.4 shows the statistical data for the deviation between M_1 and R_1. The result clearly shows the benifit of having collision count consistency algorithms. The control shows two times more deviation error than the Post-collision and the Motion-lock protocols. The Motion-Lock without the Spatial-temporal bucket synch shows the least amount of deviation for R_1 among the protocols. R_1 in the Motion-Lock with the synchronization algorithm deviates slightly more than the protocol without

the synchronization. As observed in our qualitative analysis, by manipulating colli-sion time the spatial-temporal bucket synchronization may produce more deviation. Therefore, spatial-temporal bucket synchronization reduces observed object penetra-tions, but creates more distance error. Both versions of the Motion-lock protocols show less deviation than the Control and Post-collision protocols.

Table 5.5 shows the collision inconsistency interval for the master-replica collisions between M_1 and the seven replicas on A. The Control is omitted since it does not have consistency techniques implemented and the interval can thus grow unbounded. The average interval among the three protocols shows no significant difference. However, Post-Collision protocol shows larger standard deviation and maximum interval. This indicates that by sending collision counters early before the collisions, motion-lock protocols are able to reduce the collision inconsistency interval.

Tables 5.6 and 5.7 show the amount of data in kilobytes sent and received by station A. The data are sampled at every 20ms for the entire 20min run, and the data is averaged by number of samples. The overall amount of data sent is small and the average changes are minimal for all protocols—-the collision consistency protocols do not use much more bandwidth to enable collision count consistency. We do note that the Motion-lock protocols shows larger maximum kBytes sent and received. This can be cause by the increase in data transmission when there are many collisions happening at the same time. However, on average, the motion-lock protocols do not show much more bandwidth usage.

For the two Motion-Lock protocols, with our default 100ms T_{lock} threshold, around 4% of the commands to change M_1's motion are discarded due to motion-locking. In the Motion-Lock protocol 20min run, station A generates 79074 commands to change the motion of M_1 so that M_1 turns around to move towards the center again. Out of these commands, only 3196 commands (4.04%) are discarded due to motion lock. Similarly, in the Motion-Lock protocol with spatial-temporal bucket synchronization, 3002 out of 72645 (4.13%) commands are discarded. Statistically, the amount of commands being discarded seems low and players should not feel any significant loss of control of their objects. For future work, the motion-lock protocol should be tested with real players surveying actual user experience.

Table 5.4: Deviation error of M_1 in Multi-Object Collision scenario

Deviation (pixels)	Sum	Avg.	Std.	Max.
Control	30740	0.51	2.19	42.57
Post-Collision	19342	0.32	0.86	12.82
Motion-Lock	16148	0.27	0.69	13.30
Motion-Lock w/ S-T Sync	17538	0.29	0.65	13.08

Table 5.5: Collision inconsistency interval of M_1 in Multi-Object Collision scenario

Collision Inconsistency Interval (ms)	Avg.	Std.	Max.
Post-Collision	8.52	19.88	428
Motion-Lock	7.62	4.64	39
Motion-Lock w/ S-T Sync	8.00	4.86	53

Table 5.6: kBytes sent from station A in Multi-Object Collision scenario

Send (kByte)	Avg.	Std.	Max.
Control	8.14	4.75	29.71
Post-Collision	8.08	4.68	29.71
Motion-Lock	8.09	5.01	48.83
Motion-Lock w/ S-T Sync	8.36	5.14	52.69

Table 5.7: kBytes received by station A in Multi-Object Collision scenario

Received (kByte)	Avg.	Std.	Max.
Control	27.27	9.44	61.89
Post-Collision	28.70	9.16	65.97
Motion-Lock	29.18	9.62	77.10
Motion-Lock w/ S-T Sync	29.60	10.10	79.74

5.5 Conclusion

From the analysis, we can see that the Motion-Lock protocol shows good results comparing to the standard dead reckoning protocol. Visually, by reducing the inconsistency interval, the Motion-Lock protocol reduces the collision gap created by the delayed collision resolution. With the post-collision trajectory agreement, the protocol is able to eliminate most large correction jump. With the Spatial-temporal bucket synchronization, the motion lock protocol is able to send the collision count early and display better visual result in multi-object collision.

Statistically, the use of exchanging collision counters in Post-Collision and Motion-Lock protocol helps reduce the overall deviation error. In the offline simulation analysis, comparing to the Control, the Motion-Lock protocol reduces the post-collision deviation error by half. Similarly, in the online simulation analysis, even under complex multi-object collision scenarios, the Motion-Lock protocols with and without the Spatial-temporal synchronization result in less deviation. Furthermore, the bandwidth analysis shows that on average, the Motion-Lock protocol do not require much more bandwidth usage. Therefore, the results shown in this chapter suggest that multiplayer online games with distributed architecture can benefit from the motion-lock protocol.

Chapter 6
Conclusion

In order for multiplayer online games to break away from client-server architectures and be implemented in more scalable and fault-tolerant distributed architectures, consistency for the states of player-controlled objects is required. In distributed, virtual simulations, many algorithms such as dead-reckoning, local lag, and time warp have been designed to improve consistency of game or object states in general. There has, however, not been much research specifically on consistency in collision detection, yet the majority of computer games require collision detection to display proper visual results and, more importantly, determine critical game events such as the winner of the game. Without proper collision detection, some games become unplayable; therefore, strong consistency in collision detection and resolution is essential.

In chapter 3 we analyzed a basic dead-reckoning algorithm and pointed out the problem of extrapolation error. When two objects are moving towards each other, inaccurate extrapolations can cause stations to detect different collision points, and thus, different post-collision trajectories. Furthermore, at high network latency, the extrapolation can be large enough to cause false and missed collisions. These problems create inconsistency in collision and cause the motion states to diverge. Furthermore, in the presence of network latency, jitter and packet loss, achieving perfect consistency and correctness in a distributed architecture is very difficult. We therefore defined ΔP-State Correctness, Δt_{col}-Collision Count Consistency, and $\Delta \theta_{col}$-post-collision State Correctness so that some form of consistency and correctness can be

reached.

In chapter 4, we present the motion-lock protocol that maintains consistency and improves the visual result for distributed collision detection and resolution. The protocol consists of four parts. Additional collision counters are exchanged between stations to ensure the numbers of detected collisions are consistent. The motion locking allows collision counters to be sent early before the objects collide so that the collision inconsistency interval is minimized. The post-collision trajectory agreement reduces large correction jumps for deviated objects, improving the visual result after collisions. The spatial-temporal bucket synchronization further improves the visual result for multi-object collisions by grouping closely located collision points and resolving them as one multi-object collision.

In chapter 5, the motion-lock protocol is analyzed using our offline and online simulators. The offline simulator can adjust the simulated network latency and packet loss rate to test the protocol under various network conditions. Building on top of Quazal's NetZ middleware for multiplayer games, the online simulator analyzes the protocol and measures bandwidth usage in a real network. Besides the motion-lock protocol, the dead reckoning, super-node, and post-collision protocols are also implemented and tested to compare the result with the motion-lock protocol. Scenarios are designed to unit test the protocols in different object behaviours and game environments. Collision count, inconsistency interval, deviation, and bandwidth usage are measured for each protocol. The experimental results show that the motion-lock protocol maintains consistent collision count and greatly improves post-collision trajectories deviation and visual results. Furthermore, the motion-lock protocol does not require much more bandwidth and reduces by only around 4 percent the responsiveness of player controls. As to be expected, however, complex multi-object scenarios remain challenging, and the motion-lock protocol shows less improvement as conditions become more extreme for collision detection. We also find the spatial-temporal bucket synchronization does not yield a better result in quantitative analysis, although it does provide a visually acceptable solution for collisions between locked master and unlocked replicas.

6.1 Future Work

Our Motion-Lock protocol is one of the first protocols for improving consistency in distributed collision detection. Of course, and as seen in the analysis, the Motion-Lock protocol is far from perfect. In the following sections, improvements and future work to the motion-lock protocol and distributed collision detection will be discussed.

6.1.1 Collision Count Correctness

For games with highly dynamic object, such as first person shooters, motion of the masters can be highly unpredictable, and thus increase the number of false collisions. If the stations do not filter out the false collisions, objects can be colliding constantly, which may cause the game to be unplayable. The post-collision and motion-lock protocols maintain collision count consistency, but not collision count correctness. In the post-collision protocol, stations send out collision counts for all detected collisions including false collisions, because the stations do not have enough information to discern the false collisions from the real. The receiving stations then optimistically agree to all informed collisions. In the motion-lock protocol, the potential collision detection algorithm uses linear extrapolation to estimate collisions. The estimation can be wrong, forcing objects to engage in unintentional collisions.

One possible way of filtering out incorrect collisions is to determine if the extrapolated states of the replicas are accurate or not. If a master collides with a replica with an inaccurate state, the chance that the collision is a false collision is high. The station can then steer away the replicas to avoid the possible incorrect collision. The accuracy of the extrapolated state of the replica can be approximated using the previously received states and the time gap between the states. If the past states do not closely follow the motion model, the chances are the motion of the master is unpredictable and thus the predicted motion of the replica is inaccurate. On the other hand, if the past states show constant acceleration, the replica's extrapolated state should be accurate. An additional factor is the time between updates. If the

101

gaps between correctly received states are large, perhaps caused by a congested network, the predicted motions of the replicas will themselves be based on previously extrapolated data, compounding any error. If, however, time gaps are short, the maximal deviation of a replica is more limited, heuristically suggesting better tracking of the master motion. This kind of information can be exploited to help provide cheap confidence heuristics that reduce false collisions.

6.1.2   Post-Collision Trajectory Agreement

In asynchronous networks, for stations to agree on post-collision trajectories is quite difficult. In chapter 3, the motion-lock protocol allows stations to send the pre-calculated post-collision trajectory to the other collision participating station. If the receiving station missed the collision, it simply uses the received trajectory to display the collision resolution. However, this only benefits the situation when one station misses the collision. If both stations detected the collision and calculate different sets of collision trajectories, choosing which set of trajectories to be used on both stations would require further agreement. However, any more messages passing to reach an agreement would delay the resolution of the trajectories. Furthermore, as we seen in the pre-collision agreement protocol, reaching an agreement in asynchronous network by passing messages is difficult.

Heuristically, using available local information to adaptively prioritizing which set of trajectories to be used may provide a faster solution to reach a better agreement. Local information such as the accuracy of extrapolated states of the replica, as discussed in previous section, can provide a score to each set of trajectories. By sending the score with the trajectories to the participating stations, stations can compare the scores between the local and received trajectories to determine which set of trajectories to be used.

6.1.3 Multi-Object Collisions

Designing a protocol for multi-object collisions can be difficult because of the increased object dynamics and interactions. From our experimental results, we see that spatial-temporal synchronization does not yield a better statistical result. Both versions of motion-lock show a reduced deviation, but the synchronization creates a larger collision inconsistency interval. Reducing the collision inconsistency interval reduces replica deviations, but reducing deviations may not reduce the collision inconsistency interval. The trade-offs here are interesting and worth further consideration.

Of course game development often focuses more on good visual results than precise collisions with correct physics. Therefore, instead of developing a protocol with accurate results, heuristic solutions are preferred. The spatial-temporal bucket synchronization provides a better visual result by grouping colliding objects into one collision; this avoids situations where the congested nature of many collisions otherwise causes collisions to be missed and subsequent object penetrations. Another possible heuristic solution for multi-object collisions is to calculate and send data only for collisions that are perceivable by the players. For example, when an object traveling with high velocity collides with a cluster of objects, the collisions inside the cluster may be too fast for the players to observe any visual anomalies. The post-collision trajectories for each object in the cluster that bounces outward, however, are highly perceivable. Therefore, before the fast-approaching object enters the cluster, we can heuristically pre-calculate the post-collisions trajectories for all objects in the cluster and send the trajectories to the other stations. This should improve the consistency of the post-collision trajectory and reduce the number of large correction jumps.

6.1.4 Cheating and Security

The Motion-Lock protocol is aimed at providing strong consistency for distributed architectures. Practical use in a multiplayer system also requires consideration of game security. This makes our design vulnerable to game cheating; users can maliciously craft network packets to manipulate the motion-lock protocol to force objects

to collide or not. Future work thus includes considering the use of cryptographic techniques to hide and validate the data transmitted. One other possible way of avoiding cheating is to use a trust system for the clients [13]. For the Motion-Lock protocol, clients with higher trust values would have more authority in deciding the result of a collision.

6.1.5 Fault Tolerance

In a real network, stations can crash and drop out from the network. When a station is dropped, the master no longer exists, and, before the crash is discovered by the other stations, the faulty replicas residing on other stations may not yet been removed. The faulty replicas can still collide with the other objects causing false collisions. In most cases, however, the impact on Motion-Lock is limited, since the protocol acts as an agreement only between the collision participating stations. If a master collides with a faulty replica, collision counters would be sent to inform the faulty station. Since the station no long exists, the protocol ends and the correct station resolves the collision as usual. Other stations continue to observe the collision as a replica-replica collision. Further investigation on the impact of faulty stations on the Motion-Lock protocol is of course required to verify our claims and establish the limits of fault tolerance.

Bibliography

[1] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*, pages 155–168, Berkeley, CA, USA, 2006. USENIX Association.

[2] IEEE-SA Standards Board. *IEEE standard for distributed interactive simulation - application protocols*. IEEE, 1995. IEEE Std 1278.1-1995.

[3] Jean-Sébastien Bolduc and Hans Vangheluwe. A modelling and simulation package for classical hierarchical DEVS. Technical Report MSDL-TR-2001-01, McGill University, 2001.

[4] Wentong Cai, Francis B. S. Lee, and L. Chen. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 82–89, Washington, DC, USA, 1999. IEEE Computer Society.

[5] J. Calvin, A. Dickens, B. Gaines, P. Metzger, D. Miller, and D. Owen. The simnet virtual world architecture. In *Virtual Reality Annual International Symposium (VRAIS '93)*, pages 450–455. IEEE, 1993.

[6] Addison Chan, Rynson W. H. Lau, and Beatrice Ng. Motion prediction for caching and prefetching in mouse-driven DVE navigation. *ACM Trans. Internet Technol.*, 5(1):70–91, 2005.

[7] Luther Chan, James Yong, Jiaqiang Bai, Ben Leong, and Raymond Tan. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In *NetGames '07*, pages 37–42, New York, NY, USA, 2007. ACM.

[8] Eric Cronin, Burton Filstrup, Anthony R. Kurc, and Sugih Jamin. An efficient synchronization mechanism for mirrored game architectures. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 67–73, New York, NY, USA, 2002. ACM.

[9] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet. *Network, IEEE*, 13(4):6–15, Jul/Aug 1999.

[10] T.P. Duncan and D. Gracanin. Pre-reckoning algorithm for distributed virtual environments. In *Proceedings of the 2003 Winter Simulation Conference*, volume 2, pages 1086–1093 vol.2, Dec. 2003.

[11] Blizzard Entertainment. Diablo II. `http://classic.battle.net/diablo2exp/faq/multiplayer.shtml`.

[12] Blizzard Entertainment. World of warcraft. `http://www.worldofwarcraft.com/index.xml`.

[13] Josh Goodman and Clark Verbrugge. A peer auditing scheme for cheat detection in MMOGs. In *NetGames 2008: 7th Workshop on Network & System Support for Games*, Worcester, MA, USA, oct 2008.

[14] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *NetGames '04*, pages 116–120, New York, NY, USA, 2004. ACM.

[15] Alan Kenny, Séamus Mcloone, and Tomás Ward. Controlling entity state updates to maintain remote consistency within a distributed interactive application. *ACM Trans. Internet Technol.*, 9(4):1–25, 2009.

[16] B. Knutsson, Honghui Lu, Wei Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages –107, March 2004.

[17] A. Krumm-Heller and S. Taylor. Using determinism to improve the accuracy of dead reckoning algorithms. In *in Proc. of Simulation Technologies and Training Conference*, 2000.

[18] Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

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

[20] Martin Mauve. How to keep a dead man from shooting. In *IDMS '00: Proceedings of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 199–204, London, UK, 2000. Springer-Verlag.

[21] Jan Ohlenburg. Improving collision detection in distributed virtual environments by adaptive collision prediction tracking. In *VR '04: Proceedings of the IEEE Virtual Reality 2004*, page 83, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Carol O'Sullivan and John Dingliana. Collisions and perception. *ACM Trans. Graph.*, 20(3):151–168, 2001.

[23] Lothar Pantel and Lars 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.

[24] Quazal. *Combating Latency - Game Coherence Over the Internet, Network Problems - Quazal's Solutions*, January 2007.

[25] Quazal. *Duplication Spaces^{TM}*, March 2008.

[26] Quazal. *Quazal Net-Z^{TM} 2008 Technical Overview*, March 2008.

[27] Dave Roberts, Rob Aspin, Damien Marshall, Seamus Mcloone, Declan Delaney, and Tomas Ward. Bounding inconsistency using a novel threshold metric for dead reckoning update packet generation. *Simulation*, 84(5):239–256, 2008.

[28] Sandeep K. Singhal and David R. Cheriton. Using a position history-based protocol for distributed object visualization. Technical Report STAN-CS-TR-94-1505, Stanford University, Stanford, CA, USA, 1994.

[29] Aaron St. John and Brian Neil Levine. Supporting p2p gaming when players have heterogeneous resources. In *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*, pages 1–6, New York, NY, USA, 2005. ACM.

[30] Quazal Technologies. Netz. `http://www.quazal.com`.

[31] A. Tumanov, R. Allison, and W. Stuerzlinger. Variability-aware latency amelioration in distributed environments. In *Virtual Reality Conference, 2007. VR '07. IEEE*, pages 123–130, March 2007.

[32] Greg Welch and Gary Bishop. An introduction to the Kalman filter. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

[33] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000.