

# Testing Methodologies for a Multiplayer Game Protocol

Sorin Badila, Cale Campbell, Ryan Wilk  
*Department of Computer Science and Engineering*

*Oakland University*

Rochester, United States

sfbadila@oakland.edu, ccampbell5@oakland.edu, rmwilk@oakland.edu

**Abstract**—Distributed interactive multi-body simulations are an increasingly prevalent breed of software and demand unique strategies with respect to testing. Classically non-networked multiplayer video gaming takes place on a single machine hosting a single local environment within which all players directly control actors in a simulation. Modern networked gaming often requires that a singular environment be remotely hosted with all player-controlled actors and their interactions be distributed to connected client applications. With all these simulations, it is paramount that the individual units in the simulations as well as all the moving parts in the server are rigorously tested. The research gathered for this paper will outline the testing methodologies that we find to be the most significant. Combining strong unit and integration testing inside the server, each simulation needs to be tested for usability, compatibility, and reliability. Likewise, since the integrity of the simulation is paramount, susceptibility to malicious or incorrect information fed into the system must be mitigated, and as such, we will explore mechanisms by which test the system.

**Index Terms**—Testing, Nodejs, networking, multiplayer, Web-Socket, software engineering

## I. INTRODUCTION

In our work we aim to create a testing framework for centralized simulations serving one or more clients concurrently. In order to accomplish the goal of serving multiple clients concurrently and in a timely manner, the server must communicate with the clients in an asynchronous manner in that there can be no reliance on confirming whether a data packet was received successfully, the server simply broadcasts any updates to the simulation state to all clients. With these considerations in mind, any rigorous and complete testing must account for these features. We will be introducing a number of different approaches with which to build a complete, end to end testing framework which can be used to ensure the correctness and completeness of any system utilizing a centralized simulation.

Unit testing can be achieved through traditional means of subjecting applicable functions to predetermined inputs and comparing their results to expected results. Functional testing presents a less straight-forward solution. For example, instead of treating testing in the traditional sense of invoking one method at a time, we can designate a certain action within the game to be the target of functional testing. Taking it one step further, we have the end-goal of automating such a process such that it can be invoked with little operator input,

akin to how a set of unit tests would function. In order to accomplish the task of automation, we will provide solutions to the problem of asynchronicity.

In the latter half of the paper, we will be introducing an actual instance of such a simulation, a browser-based online multiplayer javascript application called NodeTank. This application involves two primary components, a server, and one or more connected clients. Each client instance renders the game environment to the players as well as a tank object which accepts control inputs from a player. Client instances are responsible for forwarding control inputs to the server. The server is responsible for tracking and maintaining state information relevant to the gameplay. Various examples of state include health status, position, and orientation. This information needs to be forwarded from the server to the client applications with minimal latency in order to provide a continuous stream of snapshots of the game's state. Client applications are also responsible for recreating and displaying this information for the player with the end-goal of providing all players with consistent up-to-date information. We will apply the framework concepts developed in the former sections to this case study to demonstrate its efficacy.

## II. RELATED WORKS

The works of Ariurek et al[2] are very interesting to our research because they propose several mechanisms by which to introduce automated test agents into the game development cycle with the goal of finding defects. They have proposed two mechanisms by which to facilitate this automation; human-like agents and synthetic agents. A human-like agent is a separate program which learns the rules and behavior of a game via reinforcement learning. With reinforcement learning, this type of agent would learn how a human would play the game as it would have the same reward incentive as a human player, and is thus likely to detect defects which are similar in nature to those detected by humans. Their proposed synthetic agent is also a type of program which is trained via reinforcement learning, except its goals are not inline with the goals of a real human player. For example, a synthetic agent could be rewarded with implementing a scenario which would be detrimental to winning the game, but which would be likely to reveal a defect otherwise hidden from expected behavior. Using both of these methods, Ariurek et al have

created a system in which the quality of a game could be tested automatically and not in a predetermined fashion.

Rezin et al[3] developed a model checking mechanism for a specific multiplayer game. They did so by creating a list of attributes which are mapped to parameters into the model - for example, each object must have some position identifier, X/Y/Z as well as a vector which describes the orientation. Our case study, NodeTank, will also suffer from the same problem as their case study in that state explosion due the millions of possible position/orientation combinations and as such the game model must be reduced to meaningfully study it.

Peusaari et al[5] discuss the computational issues and challenges of distributed human-in-the-loop simulations of a basic architecture consisting of several satellite components focused around a management component. The specific components include a client, server, motion platform controller, I/O controller, and a manager. The manager distributes setup instructions for the simulation as well as collecting and processing data streams from the other components. The servers play the primary roles of computational units performing physics/dynamics processing. The relationship between these client and server components are analogous to the client-server relationship NodeTank utilizes. The piece that we will need to construct is the manager, a component that will allow for the distributed initialization of tests and the data collection of those tests. However, in our case, this manager will observe and report on the behavior code itself rather than sensor data. Components that don't translate to our work are, with reason, the motion controller. Several of the challenges of distributed simulation that are relevant in this paper may be relevant to our work as well. Peusaari outlines the following three main challenges to the distributed simulation. The end result of the simulation should be capable of executing in real-time. Secondly, the system, being distributed across a network, will be naturally intolerant of delays. The more delay that is introduced, the more the data and validity of the experiment drift. Thirdly data transmissions should be well-planned and organized in such a way that minimizes hindrance of the simulation and its core goals. Multi-body simulations require that, at a minimum, coordinates and orientations of bodies subject to physics and dynamics calculations be routinely transmitted at reliable intervals.

These are all concerns of NodeTank. While they may be to a lesser degree, as NodeTank is a game rather than a tool for executing experiments for research, they will be valid concerns to the degree of their perceptibility. As delays grow, corresponds to the players' abilities to enjoy the experience decline.

### III. APPROACH

Our approach is to develop a testing framework for NodeTank will combine abstract testing of models, unit testing, and functional testing into a single, yet modular, utility. The effort of functional testing will involve the synchronization of timing of outputs from the server as well as the clients involved. The testing utility should be distributable just as

the software being tested and a means of distributing and launching a test should be runnable from a single test location. Any data and any results of a test run should also be collected and delivered to the single test location where it will be analyzed and classified as either a success or failure. Current design will feature the integration of the Labstreaminglayer tool and a Labstreaminglayer server. This tool will allow for the collection of any data or output deemed necessary for a given test. It will allow for record sub-millisecond timing of events from multiple machines over a local area network.

#### A. Network Architecture

In attempting to understand the architecture of a centralized simulation, we must also understand under which circumstances such an architecture is desired. For example, let's assume that we have some data that we would like to share between one or more clients, with a client being a receiver which is interested in some data. At a very high level, such a data sharing layout can be split into two categories: peer-peer and client-server. Smed et. al[6] describes the different layouts in detail, however, generally a peer-peer architecture is defined as having two or more clients which have all, or part of some data which is then shared equally each peer(Fig. 1). In this scenario, each client is equal to every other client, and thus, would have to be fully connected to one another[6]. This type of architecture has its own advantages when it comes to certain types of systems, such as file sharing and lock-step simulations, however we cannot maintain the type of centralized simulation which is pertinent to our topic.

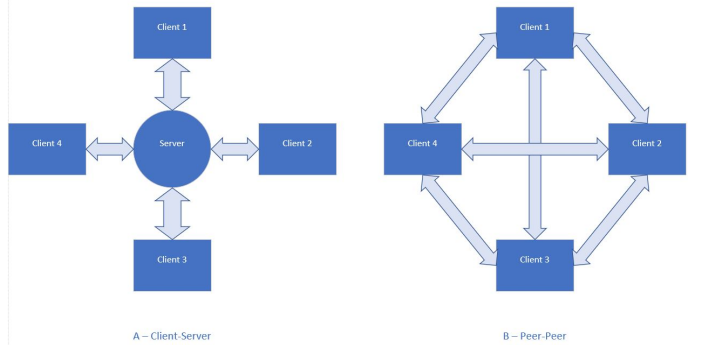


Fig. 1. Example of a peer to peer and client server architecture

Another type of network architecture exists which has one server serve multiple clients, client-server(Fig. 1). In this type of architecture, one node is designated as the server, and all authoritative communication is handled through it before being sent to the clients[6]. As noted in Fig 1, we can see that any number of clients may connect to the server at any one time, and clients do not have to be fully connected since the server is the authoritative source. The client-server architecture will be the target of our testing methodologies and framework. Although peer-peer has its own valid use cases, it is not appropriate for a centralized simulation and would likewise necessitate a different approach to testing.

### B. 3D Rendering at a High Level

In the real world, we have object which we can look at and interact with. When such objects are created in a 3D renderer, some data must be modeled about the object. For simplicity, we will only consider a Cartesian coordinate system(Fig 2), in which points are mapped by numerical coordinates along three perpendicular lines(axes). Intuitively, the positional information may be expressed in terms of a displacement along each one of the three axes (X, Y, Z). As such, a simple vector with three components can encode this information(Fig 1.):

$$(V_x, V_y, V_z)$$

Fig. 2. Vector Used to Encode Position

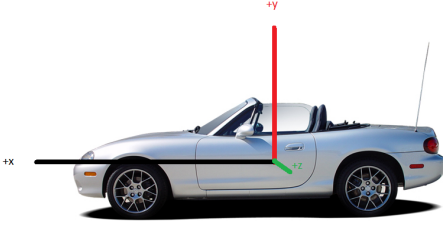


Fig. 3. Example of WebGL Coordinate System

However, simply listing the position is not sufficient in order to accurately map the object in space. An object may be rotated around its position. Fig 2. shows a simple example of a car being rotated about the Z axis. The simplest method by which this information can be encoded is to use another three component vector to keep track of the rotation around each axis, and then apply each rotation to each axis in a cascading fashion. This is commonly known as an Euler Angle, which is intuitive to use, but not sufficient enough in our use case due to the possibility of gimbal lock. In short terms, gimbal lock can occur when two axes are in a parallel configuration to each other, which would force an otherwise single axis rotation to instead become a composite rotation.

The conventional solution to this problem is to discard the Euler Angle representation of rotation in favor of using quaternions. For the purpose of this paper, we must only know that an equivalent description of orientation can be encoded in a quaternion, but without the gimbal lock limitation which would cause rotation behavior should it occur. In short, this quaternion may be expressed as a four dimensional vector, similar to the one expressed in Fig 1., but with an extra component,  $w$ .

At this point, we have all of the necessary information to encode a static object in our game world, yet there is one more

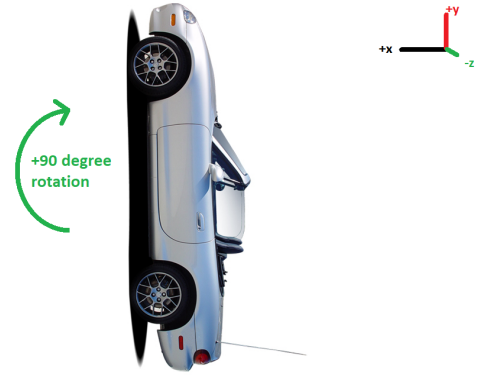


Fig. 4. Example of Rotation Around the Z Axis

attribute which must be implemented: velocity. Consider that an object in our game is in motion, assume it is moving along the X axis. If we have enough snapshots of its location, then we may be able to render its path up until the current snapshot. However, consider what happens until the next snapshot is received; we will have no idea about the object's path. In order to circumvent this, information about how fast the object is moving may be encoded such that an extrapolation between snapshots may occur. This velocity attribute is split into two segments; the linear velocity, which is the change in position across the Cartesian system, and angular velocity, which is the change in orientation. Each one of these attributes may be encoded in a three dimensional vector(Fig 1).

$$\text{Position:}(V_x, V_y, V_z) \quad (1)$$

$$\text{Orientation:}(V_x, V_y, V_z, V_w) \quad (2)$$

$$\text{Linear Velocity:}(V_x, V_y, V_z) \quad (3)$$

$$\text{Angular Velocity:}(V_x, V_y, V_z) \quad (4)$$

Fig. 5. Complete Object State

Fig 6 shows our final object state. This will be our basic building block when we will discuss the protocol in detail, as this will become the data packet which updates clients on the state of the simulation and the objects therein.

### C. Game Model

The protocol workflow has been described at a very high level, but it is agnostic to the specific game and the rules associated with it. The player is modeled by a tank object. This tank object has a few properties associated with it: alive, outOfBounds, score. The alive state dictates whether the tank is on the playing field - given that in our rules we have established that being shot simply leads to a respawn, this state is only used to indicate that a respawn must occur and that the other player's score is incremented. The outOfBounds attribute is true when the player steps outside of the game field, which leads to alive being set to false and score being

decremented. The score attribute keeps track of the player's current score.

A keen observer would note that a player's tank has many more attributes than those listed above. Rezin et al [3] utilized model checking on a multiplayer game, and they came up with an attribute list which contained all variables that would change over the course of the game. The attributes are tied into parameters, which are constants set at the beginning of the game. For example, a player's tank can be modeled as a parameter with attributes X position, Y position, Z position, lookAt, score, alive, outOfBounds. The limitation of using this approach is that if an object's position is utilized in checking the model, the list of all possible state combinations would be too large to ever compute due to the size of the game field having granularity in the tens of millions and the total number of possible lookAt locations also being in the millions.

With these considerations in mind, model checking has also been overlooked in favor of utilizing automated testing to test the game directly for consistency in its rules. One thing of note is that the formal definition of the game rules for example, a valid x coordinate, is syntactically equivalent to the check the game logic would perform; therefore writing a model checking program is redundant in this specific case. Fig 11 shows the formal definition along with the implementation of the rule.

$$(W_{xmin} \leq T_x \leq W_{xmax})$$

$$!(playerTanks[k].obj.position.x < xMin ||$$

$$playerTanks[k].obj.position.x > xMax)$$

Fig. 6. Formal Rule and Implementation of Valid X coordinate

#### IV. IMPLEMENTATION

With the all of the prerequisite work complete, the implementation of the the game and protocol will be discussed. For reference, the project code is located at (<https://github.com/spac3nerd/CSI-5390Proj>). The nodejs server will need to run both traditional HTTP transactions and socket transactions. The HTTP transactions are used to ask the server if there are any open spots in the game, and whether access is granted to a new player. If access is granted, then the server will return an unique token to the client with which a socket connection can be requested. The client then requests entry into the game with the token and a tunnel is established between the client and the server.

In the following sections we will cover the specific testing implementations with the goal of addressing the problem of rigorously testing NodeTank, or any other multiplayer game built on the NodeJS platform.

##### A. Automated Functional Testing

The first testing scenario we will consider is to allow the system to automatically play the game in order to test a set of pre-defined set of actions. For example, we can

programmatically interact with the game directly in order to control one of the player tanks. With the system having control over a player character, we can individually test the different aspects of the game: movement, shooting, respawning etc... This type of automated testing could be invoked on an individual developer's machine in order to test each function of the game as its being developed.

However, we may take this testing even further and consider that since the game runs within the browser, we would be able to tap into any concurrent client simulation. This has the implication that we would be able to inspect the state of each client concurrently.

##### B. Automated Unit Testing

To be developed

##### C. Remote Testing

1) *Core RPC Services*: Remote testing of the target application is achieved through an adaptable and extensible pair of RPC services. Two APIs, 'Data' and 'RemoteCtrl', that enable remote testing and they are capable of being updated and expanded quite easily as the application's range of behaviors grow during its lifetime. Each API service is defined in an IDL(Interface Description Language) file. Each defines the a service's name followed by generic a list call signatures of the functions that should be exposed by the service. The IDL offers several plain data types such booleans, floating points, a string type, and signed integers of the typical varying bit-widths. Basic C-style structures and enumerations may also be defined to add versatility to the services defined types.

2) *RPC Adaptability*: As the application grows and new types of interactions and information need to be handled, the same pair of IDL need only minor editing by the maintainer. Automatic generations scripts may be rerun at each stage of changing demand. The resulting output is a fully-formed supporting library in the target programming language of choice for, both, the client and server sides of each RPC service. The RPC-server half of each API is included and/or linked to from the simulation's server to allow both services awareness of and access to all necessary data of the simulation's clients. In the event of new or changing data types, the client simulation needs only to be rebuilt in the case of a compiled-language and no changes are required for interpreted languages. In the case of new or changing functions, the auto-generated RPC handlers need only be given function bodies by the tester. Building remote testing utilities for the simulation is easily done thanks to this process, as there are no dependencies between the programming language of the simulation's server and the RPC client. The RPC service itself manages all translation between native data types of the server and it's RPC client. This means the RPC service manages the serialization/deserialization and transport of all function arguments and return values, byte-order translation between differing host hardware, and member alignment between differing host operating environments. All these features allow for a simulation written in javascript, C++, Java, Python, C#, etc. to agnostically cooperate with any client

written in any other supported language and host operating environment.

3) *Distributability*: With an underlying TCP or UDP transport layer, this remote testing strategy allows for a wide range of geographical testing configurations to be thoroughly explored. Testing and examination of the system may be performed in a close-proximity LAN configuration, or the simulation server and its clients may be much more geographically separated, communicating over the Internet with one another and the testing client itself.

## V. CASE STUDY

### A. Remote RPC Services

Our implementation of the *Data* and *RemoteCtrl* APIs were built upon a tool known as Thrift, originally created by Facebook, but now managed by Apache. The RPC services contain several functions, between the two of them, for retrieving and executing testing and control operations including *GetTokenByName*, *GetPose*, *SetMove*, *Fire*, *StartDataServer*, *ExecuteTest*, *GetTestResults*, and *GetTestCases* to name a few. The RPC handlers of each RPC services are setup within *gameCore*, the game's primary server. From here, they are pointed to the appropriate handling functions. These handlers allow for the computational logic of the services to access all the necessary information from *gameLoop.js*, *global-state.js*, and *socket-manager.js*.

1) *Remote Testing Service*: Within *gameCore* the RPC host objects are setup each to operate within their own thread, alongside the game's primary servers.

2) *Remote Control Service*:

a) *Injecting user input*:



Fig. 7. Example of a testing application built using our remote testing APIs. This instance can cycle between multiple remote simulation servers. It populates a list of available unit tests available on that server, can execute them, and then presents the results to the tester by highlighting list items in green or red to indicate pass or fail respectively.

b) *Extracting the rendered environment*:

## VI. CONCLUSION

### A. Future Work

To be developed

## VII. REFERENCES

- [1] A. Valadares, "Aspect-oriented architectural style for distributed interactive simulations", 2016, Available from ProQuest Dissertations & Theses Global. (1881527059). Retrieved from <https://search-proquest-com.huayu.kl.oakland.edu/docview/1881527059?accountid=12924>
- [2] S. Ariyurek, "Automated Video Game Testing Using Synthetic and Human-Like Agents", 2019, Available: <https://arxiv.org/pdf/1906.00317.pdf>
- [3] R. Rezin, "Model Checking in multiplayer games development", Innopolis University, 2017, Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8432324>
- [4] R. Hofer, "DIS Today", 1995, Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=400453>
- [5] J. Peusaari, "Distributed Issues in Real-Time Interactive Simulations", Department of Technology Lappeenranta University of Technology, Finland, Available: <https://ieeexplore.ieee.org/abstract/document/5361761>
- [6] J. Smed, "Aspects of Networking in Multiplayer Computer Games", Proceedings of International Conference on Application and Development of Computer Games in the 21st Century, Hong Kong, 2001, Available: [https://www.researchgate.net/publication/269251176\\_Aspects\\_of\\_Network](https://www.researchgate.net/publication/269251176_Aspects_of_Network)