Egal Car: A Peer-to-Peer Car Racing Game Synchronized Over Named Data Networking

Zening Qu [*]

UCLA REMAP

Jeff Burke [†]

UCLA REMAP

December 26, 2012

Abstract

Multiplayer online games require synchronization mechanisms to maintain game state consistency among all the players. This project explored the use of the emerging collection synchronization primitives in Named Data Networking (NDN) to easily synchronize a simple multiplayer car racing game, Egal Car, in a peer-to-peer fashion. The game was based on a Unity game engine demo application. To implement the experiment, C# bindings usable in Unity were built for PARC's CCNx library, including the new Sync protocol. The impacts of the unordered nature of default CCNx synchronization and unreliable delivery in basic Interest/data exchange were explored. This work will inform an upcoming exploration of how to build a more extensive multiplayer role-playing game using NDN.

1 Introduction

One of the most important differences between a multiplayer online game (MOG) and its single-player counterpart is that the former requires a shared virtual world and game state consistency across network-connected players. Consistency ensures that all players in the shared virtual world are able to form a correct understanding of it and have the desired orientation to it. For example, a car racing game should provide each player with a *consistent* view of all the cars' positions in real-time, and there should be no collision between any two views.

[*]quzening@remap.ucla.edu

[†]jburke@remap.ucla.edu

For a MOG with players distributed over a wide geographic area, delivery of game state updates usually relies on packets transmitted over the Internet. This paper documents the implementation of a simple MOG over a possible future Internet architecture, Named Data Networking (NDN) [9]. In particular, it explores the use of new collection synchronization mechanisms now available in NDN. As discussed below, NDN names data instead of hosts. It has inherent multicast data delivery, caching, content signing, and collection synchronization capabilities that could provide substantial benefits for multiplayer games and distributed simulations. As it relates to these applications, NDN can be regarded as a new datagram delivery service, an alternative to TCP/IP. However, NDN also has the potential to impact game or simulation design and conceptualization, and to assist implementation and improve performance.

By creating a MOG on the NDN testbed, NDN's impact on MOG design and implementation was explored. The simple MOG described here, Egal Car, provides preliminary experience to inform the creation of a more sophisticated game and a test for C# libraries usable with the Unity game engine.

Egal Car is a 3D car racing game with a highly detailed track and scenery. Each player has control over one car, which can increase in speed, slow down and alter direction. The cars can collide with each other and with fences along the track. All car movements are simulated by the game's physics engine. Like in other racing games, each player's goal is to win the race. Egal Car is adapted from an open source game template, Unity Car Tutorial [11]. This template provides the necessary graphics and physics modules, but it is a single-player offline game. The project designed and engineered this network module, a synchronization mechanism with the needs of this car racing game in mind.

Following are descriptions of the design choices and details of the implementation. Section 2 introduces NDN principles and facts that may inspire MOG design. Sections 3 through 5 discuss MOG design, using Egal Car as a main example.

2 NDN Background

NDN retrieves data based on application-defined names rather than host addresses. To retrieve data, the consumer must know the name, instead of the data's location (host address) as in IP. NDN communication uses two packet types: *Interest* and *Data*. An Interest packet is issued by the receiver to express what set of data is needed. A Data packet is published by the sender in response to an Interest. Both Interest and Data packets use names to identify the data being exchanged (see figure 2). An Interest is "satisfied" when a Data packet is received with a Content Name that falls within the prefix of the Content Name in the Interest packet. For more detail, see [9].

This project explores higher-level features being made available in NDN for collection synchronization. There are a few general purpose synchronization protocols under development. Among them is the CCNx[1] Synchronization

[1]CCNx is an implementation of NDN and an open source project by Xerox PARC.

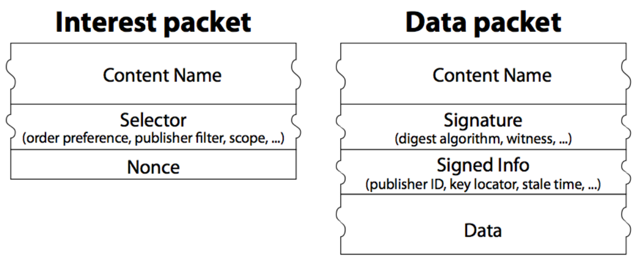
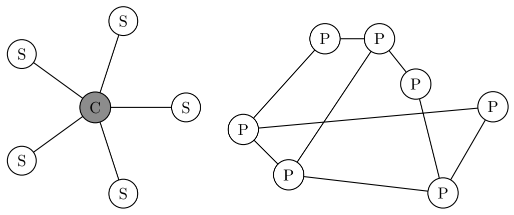

Interest packet
Content Name
Selector
(order preference, publisher filter, scope, ...)
Nonce
Data packet
Content Name
Signature
(digest algorithm, witness, ...)
Signed Info
(publisher ID, key locator, stale time, ...)
Data

Figure 1: Interest packet and Data packet



(a) C/S (b) P2P

Figure 2: C/S architecture and P2P architecture

Protocol (CCNx Sync) [3], which synchronizes collections of named data under a given name prefix. The research examines how it can be used to support game synchronization, as described above. The results show that CCNx Sync can provide significant benefits to MOG development by handling many of the low-level details for maintaining consistency across different instances of the networked game.

3 Architecture

There are two main classes of MOG network architectures, client/server (C/S) and peer-to-peer (P2P). In C/S architectures, all clients directly communicate with a central server, which is the only authorized source of game state updates (see figure 3(a)). In P2P architectures, peers (clients) communicate with each other directly, and each peer computes the latest game state on its own (see figure 3(b)). Typical implementations in major multiplayer online games appear to follow the C/S model using connectionless protocols such as UDP.[2]

Despite its rarity in implementing major multiplayer online games, it is generally agreed that P2P approaches outperform C/S in scalability, latency

[2]Connection-oriented protocols such as TCP can also be used; however, others have shown that such protocols may result in severe performance degradation. Thus, they are rarely used by MOGs [7].

and robustness [7, 13]. However, in typical IP implementations, P2P game network architectures face significant performance limitations. For example, due to limited deployment of IP multicast [7], they have higher bandwidth requirements than C/S games. A common scenario is that every player wants to exchange packets with all other players. For example, in Egal Car, every player would benefit from knowing the positions of all the other players' cars. When this is implemented by direct communications between all the players, it would lead to an enormous amount of traffic. In a C/S network, on the other hand, the traffic is alleviated because the centralized server plays the role of an "information aggregator", and each player communicates with all other players by exchanging packets with the server only.

The lack of a central authority in P2P architectures is a more fundamental limitation, making player and update authentication more difficult and cheating easier [13].

P2P architecture was chosen for this project's prototype, Egal Car. NDN makes a P2P-based multiplayer game much more feasible. Because NDN has multicast data delivery content caching built-in, it can substantially reduce network traffic in one-to-many scenarios like P2P state distribution. Game state can be obtained by many peers from one or more sources on the network using NDN's inherent multicast and caching. Furthermore, because NDN secures every data packet rather than securing the communication channel as currently done in SSL, cryptographic authentication of signed content objects can be used to alleviate the issues related to cheating.

The choice of game architecture greatly impacts other network module design decisions, such as the synchronization mechanism. For example, in C/S architectures, clients might send certain types of user inputs to the server, and the server would compute the result of the inputs. A client would not compute the result on its own because it has neither the right to compute the next game state nor the information that is sufficient to do so. Instead of claiming, "I am three blocks away from my last position now", a client would only tell the server, "I want to move three blocks to the north-east", and let the server confirm its new position. On the contrary, with P2P architectures, every peer has the right and the information to compute its own next game state, assuming all the players can maintain consistent game state. This gives more freedom to individual players in deciding what to send to other peers. A peer can choose to send out user inputs, as a client would in C/S architecture, or choose to send out the result of its inputs and ask the other peers to update their information. The later approach has been adopted in Egal Car for synchronizing information about each car's position (see section 4.2).

4 Synchronization

In this project, asset synchronization and state synchronization are approached differently to explore the requirements and constraints of slowly and quickly changing data. These two approaches may be combined in future work.

Below, (1) data are classified as either *assets* or *states*; (2) data to be synchronized are identified; and (3) different experimental approaches for asset and state synchronization are defined.

4.1 Namespace

The data were identified within a namespace design that accounts for the assets and relevant state of the single-player application used as the basis for Egal Car. According to Knutsson et al. [10], a virtual world is typically comprised of immutable elements (terrain) and mutable elements (players, non-player characters, mutable objects, mutable landscape). The immutable elements are to be installed before gameplay begins, at level changes, and other significant moments. The properties, instantiation and deletion of the mutable elements are to be synchronized in real time.

In Egal Car, there are three types of data:

1. the terrain, track, and car graphics, which are immutable, unchanging across peers and thus do not need synchronization;
2. asset creation and deletion, controlled by player login and logout; and
3. synchronized state, which includes properties of the assets as well as global game state.[3]

The time between asset updates (player login and logout, in this case) may be measured in seconds or even minutes, but the state changes more regularly and rapidly in the cases being considered. In Egal Car, a car's position, for example, tends to be calculated for every video frame. The time between a car position state update and its successor is usually measured in milliseconds. It is relatively easy to predict the desired frequency of state updates, as this is often guided by game applications frame rates and perceived interaction latency.

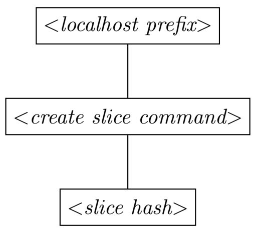
Figure 4 illustrates Egal Car's namespace, composed of three trees. Each node contains one or more name components. Each path that starts from the root denotes a name.[4]

The object tree (figure 4(c)) shows all asset and state names, in blue and red respectively. The root node is composed of the `<topological prefix>`, where game data can be published, and the game's name, `EgalCar`. Our instantiation of the application for testing used a `<topological prefix>` of `/ndn/ucla.edu/apps`. Other NDN games would use their own values for `EgalCar` and `<topological prefix>`.

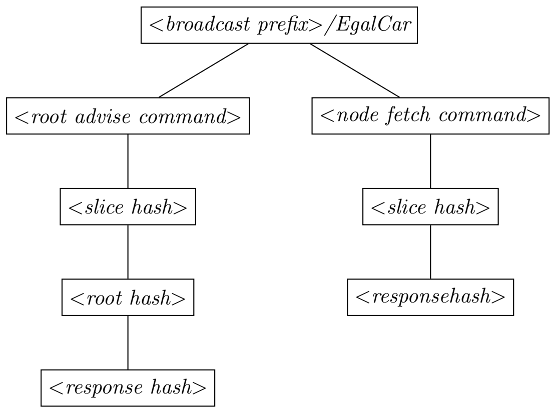
In the figure, the leftmost child of the root shows the namespace for cars in the game. Nodes using this convention (and their children) are dynamically

[3]State nodes do not come in and out of existence on their own, but are created and deleted with an asset or game. Conceptually, global game state can be considered properties of a root node.

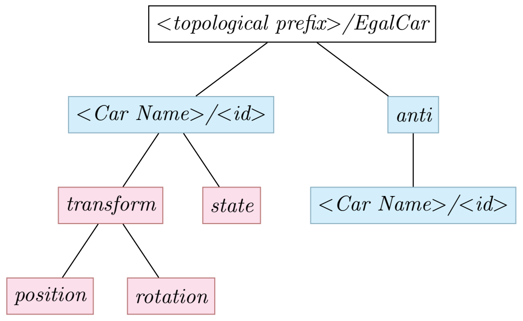
[4]The brackets `< >` around name components denote that they will be substituted with "real" values at run time. For example, when Egal Car is executing the actual name might be `/ndn/ucla.edu/apps/EgalCar/AliceCar/123456/transform/position`.



(a) Repository Tree



(b) Discovery Tree



(c) Object Tree

Figure 3: The NDN namespace of Egal Car.

created when a player joins in the game. Since Egal Car is a MOG, there will be more than one `<Car Name>/<id>` node, as players discover each other (see 4.2). The `<id>` component, which will be substituted by a random number during run-time, is designed in case there is a name collision in `<Car Name>`. The right child of the root node is used to indicate that a certain player (car) has quit or disconnected. The `anti` node and its child nodes are all assets. The remaining nodes are state names, and they are synchronized differently than assets.

As stated in Section 2, CCNx Sync can be used to aid multiplayer game development. Egal Car uses its basic features to support asset synchronization, game discovery and player discovery, given the proper namespace design. To do so, two other trees (the Repository Tree and the Discovery Tree) are needed in the namespace. The Repository Tree is a namespace used by each game peer to communicate with its local CCNx repository, where asset data is stored and synchronized by CCNx Sync. `<localhost prefix>`, the root node of the Repository Tree, confines the communication scope to localhost. The Discovery Tree is the collection of names used by CCNx Sync. Its root node is similar to that of the Game Tree except that it has a broadcast scope (`<broadcast prefix>`). Since CCNx Sync is used for "discoveries", it is important that its packets be broadcast to reach all potential data publishers that may appear anywhere in the network. More details of the Repository Tree and the Discovery Tree follow in Section 4.2.

4.2 Asset Synchronization

Asset synchronization, specifically player entry and exit, is implemented through the discovery of new objects in the sync tree. The process is discussed in some detail here as Sync is a relatively new primitive in the NDN architecture.

When an instance of the application is initialized for the first time, a *slice* is defined in the local repository [2] that defines the content objects to be synchronized. At that time, two names are provided to the Sync mechanism: the `<broadcast prefix>` and the `<topological prefix>`. The first defines CCNx Sync's broadcast namespace to use in communicating with other Sync daemons, while the second is the common prefix of every name in the slice.

An application informs its local repository about the creation of a slice by issuing a start write Interest for the slice configuration object. The name used in this process is `<localhost prefix>/<create slice command>/<slice hash>`. `<localhost prefix>` confines the packet within the same node; `<create slice command>` claims the command type; `<slice hash>` is the SHA-256 hash of the slice configuration object and is used as the name of the slice in the Discovery Tree.

Once the slice is created, CCNx Sync broadcasts *Root Advise* Interests periodically in the `<broadcast prefix>`. Root Advise Interests have names of the form `/<root advise command>/<slice hash>/<root hash>`. The `<slice hash>` identifies the particular collection of names to be synchronized. The `<root hash>` is a hashed digest of a peer's slice.

So, to participate in a given game, an Egal peer will have that game's `<slice`

`hash>` in its Root Advise Interest. Other peers running the same application recognize that peer because they share a `<slice hash>`.

A peer's collection of game assets is represented by its `<root hash>`. If this `<root hash>` is the same as that of everyone else's `<root hash>`, then all peers are in sync. If it is not the same, the other peers would respond to the peer's Root Advise Interest with a Data packet named `/<root advise command>/<slice hash>/<root hash>/<response hash>`. CCNx Sync uses *Node Fetch* command and normal Interest/Data packet exchange to then reconcile differences in the collection among peers, updating the repositories to be consistent.

To announce the creation of a new player (car) to the network, a content object named `<topological prefix>/EgalCar/<Car Name>/<id>` is published by the player's game instance into its local repository. By using `<topological prefix>`, this content object becomes part of the slice managed by CCNx Sync. The data of this content object are some configuration information of the car (initial position, 3D model's name, player information, etc.). As other peers learn about the newcomer through CCNx Sync, they use the configuration data to instantiate a 3D car model in the given position, and they mark it with the given player information.

When a player quits, it writes an *anti-asset* into its own repository. This provides confirmation of object deletion. (Note that the CCNx repository does not support per-object deletion.) The anti-asset's name would be `<topological prefix>/EgalCar/anti/<Car Name>/<id>`, in which the `<Car Name>` and `<id>` represent the car that belongs to the quitting player. With CCNx Sync, synchronizing anti-assets is the same as synchronizing assets. Peers that learn about the anti-asset destroy the corresponding avatar in their game world. If a player is forced to quit, due to network failure for example, there is no chance to write the anti-asset. In this case, the anti-asset would be written by other peers who remain connected.

4.2.1 Unordered, reliable synchronization

CCNx Sync provides unordered data synchronization, i.e. it may discover assets that are created later before discovering assets that are created earlier. In Egal Car, assets are independent of each other, and there is no causal relationship between asset updates as long as there is an application strategy to know when to begin the car race. Thus CCNx Sync can be used as-is. Future work will consider the tradeoffs between the unordered synchronization strategy, which works in the limited case of Egal Car, and contemporary ordered approaches ([4, 1, 8, 5, 6]) for other game types. These ordered synchronization algorithms trade off delay for ordering, either holding every peer's turn to wait for the correct packet to arrive, or performing a rollback when disordering is detected. These ordered synchronization strategies will be needed for updates that are correlated, but they are not necessary in the Egal Car.

Although asset synchronization is unordered, it does need to be reliable. As an example, Alice and Bob must be known by all other peers regardless of potential packet losses. NDN, similar to IP, only provides best-effort datagram delivery. The reliability in asset synchronization is achieved by CCNx Sync, as the synchronization routine periodically broadcasts Root Advise Interests throughout the lifetime of the game. As soon as any difference between any repositories is detected, CCNx Sync will reconcile the difference.

4.3 State Synchronization

In Egal Car, state is a snapshot of a variable (property) associated with an asset or the game instance. In the application namespace, state updates are versioned content objects that are children of a uniquely named asset. For example, state updates of Alice's car use this name `.../<AliceCar>/<id>/transform/position/<version>` to refer to Alice's position, where `<version>` is a timestamp. As with asset creation and deletion described above, the states of Alice and Bob will not affect each other: `.../<AliceCar>/<id>/transform/position` updates will not cause `.../<BobCar>/<id>/transform/position` to change. Note that this does not relate to the case when Alice and Bob interact with each other. Such interactions lead to players' state change. A new model for these interactions is being developed for future work.

4.3.1 Ordered, unreliable updates

Egal Car state updates are snapshots rather than change logs. For consistency of visual rendering, they should be ordered, with the most recent state update representing the object state, and undelivered state updates ignored. State synchronization does not need to be reliable as long as there is either no player interaction (global state is deterministic based on player state input) or a facility is available to periodically confirm overall state.

Therefore unlike asset synchronization, it is unnecessary, or even undesirable, to use reliable game synchronization algorithms for state synchronization in the Egal Car scenario. When a state update is lost, there is no reason to hamper the game's pace to wait for retransmission in order to achieve an ordered state update or to perform rollbacks, as the latest state is what the gameplay needs. (Again, this is for the limited case of no permanent modification to the asset's behavior happening because of a state update.)

Instead of CCNx Sync, NDN's Interest-Data exchange is used directly for state synchronization in Egal Car. Each player's instance issues Interests for the state updates that it wants to follow, and maintains a *timestamp floor* for each state update series. For example, Alice follows Bob's position by periodically expressing an Interest in `.../<BobCar>/<id>/transform/position`. To this Interest, NDN nodes may respond with any data matching its prefix, so the Interests must be constructed to obtain the latest child for a given game object prefix. This is achieved by setting selectors in Interest packets appropriately: The *rightmost child selector* is set, and an *exclusion filter* is used to exclude data with versions less than or equal to the timestamp floor. The result is that a player may miss some state updates if a newer one has already been published, but whatever state reaches the player will be at least later than the previous one received.[5]

4.3.2 Traffic optimizations

State synchronization is what generates the largest amount of traffic in most MOGs [10]. In many cases every player wishes to follow the state of all other players in real time, and every player is expected to exchange several data per second[6] In a P2P network of n players, the number of packets being sent for state synchronization per second (N) can be estimated by $N = xf\bar{h}n(n-1)$, where x is the number of packets necessary for a state exchange, which is usually greater or equal to two. f is the frequency of state exchange defined by the game application. $\bar{h}$ is the average number of hops that packets must travel. In IP networks, $\bar{h}$ equals the average distance (in hops) between peers, as each peer is both a data receiver and a data sender. In NDN networks, $\bar{h}$ will be significantly smaller because most Interest and Data packets do not need to travel from each data publisher to all receivers. Interests for the same data aggregate on their way towards the data publisher, Data packets propagate along a multicast tree, and are cached at each hop [9]. Many Interests need to travel only a few hops to be satisfied by data from cache, or to be combined with an identical Interest. This reduces the average travel distance of packets, and should make NDN games generate much less traffic than IP games.

Some optimization can be done to the game application's packet contents to further reduce the traffic generated by state synchronization. In Egal Car, the `state` node in the Game Tree (see figure 4(c)) is designed for such optimization. The `state` object is a summary of other state updates (`transform/position` and `transform/rotation`), and is synchronized instead of its siblings.

5 Implementation

Egal Car is developed with Unity, one of the ten most prominent game engines [12]. It is adapted from Unity's single-player car race tutorial [11]. The tutorial's gameplay has been left intact, and the project's network module makes the tutorial a MOG. After implementation, the synchronization for a small number of peers (2-3) was tested, and game play was evaluated.

The network modules builds on the CCNx libraries, as illustrated in figure 5. A CCNx daemon named `ccnd` provides basic Named Data Networking support, such as sending and receiving Interest and Data packets. Asset synchronization requires a local repository, such as CCNx's `ccnr`, which manages read/write requests to the repository. Finally, CCNx Sync is used to *manage* repository contents (see section 4).

[5]Note that because any NDN node may answer an Interest with cached data, the state that is received may not be *globally* the latest state, but will at least be later than the timestamp floor given in the exclusion filter.

[6]Larger environments may incorporate some type of partitioning if there are a very large number of players.

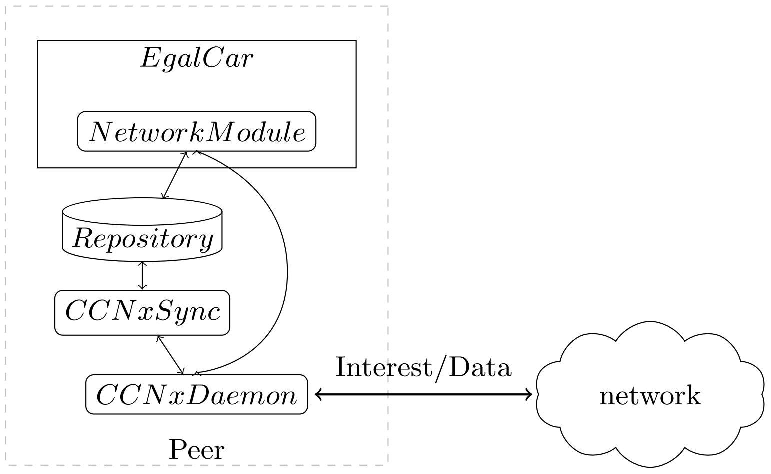


Figure 4: CCNx components used by Egal Car.

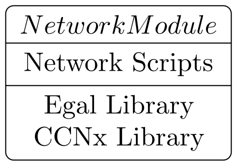


Figure 5: The network module's relationship to other libraries.

The Egal network module is where asset and state synchronization is implemented. It contains `C#` scripts for Unity and two `C` libraries (see figure 6). The network scripts implement the project's synchronization mechanism by invoking functions provided by the Egal library and the CCNx library. CCNx library contains programming interfaces given by the CCNx project. Egal library is based on CCNx library, but it provides some convenient functions for synchronization. The *InteropServices* of the *.NET Framework* are used to invoke `C` functions from `C#` and vice versa.

6 Discussions and Future Work

Egal Car is a relatively simple MOG prototype, and the project's synchronization mechanism was easy to develop using NDN collection synchronization. With Egal Car, assets are independent of each other, and they have no explicit interaction. This is why unordered or unreliable synchronization strategies can be applied to the game without hurting its consistency.

The project's synchronization mechanism does not apply to game genres that have numerous user interactions, such as role-playing games (RPG) and real-time strategy games (RTS). Inter-user interactions would lead to state changes and even asset changes. A new model is needed to describe interactions, and a new synchronization strategy would have to be designed (or chosen) to meet its requirements. Studying a new model and synchronization strategy is planned for the future.

Egal Car is not an appropriate test for scaling, as it has few asset updates and a limited number of players. To evaluate whether, or by how much, P2P games would be more scalable on NDN than on IP in general, new test cases would need to be implemented. Building a multiplayer online role-playing game with more sophisticated asset and state interaction is planned, as is the study of scaling through actual play and simulation.

Finally, of special importance to P2P games is the research of game security and cheating-prevention. Since network security in NDN is approached quite differently from that of traditional IP networks[7], numerous updates to authentication and anti-cheating mechanisms are expected.

7 Conclusions

This project developed straightforward synchronization mechanisms for simple car racing games over NDN, using the CCNx sync protocol and bindings created for the Unity game engine. The mechanism was implemented in a prototype game, *Egal Car*, and organized into an open-source synchronization library, *Egal*, to support future NDN game developers. Also explored was the workflow of NDN-based multiplayer game development. Collection synchronization approaches using named data appear to hold promise for streamlining development and performance of multiplayer online games and distributed simulation, especially in terms of simplifying development and deployment of peer-to-peer architecture. For simple games like Egal Car, where updates are independent of each other, default unordered and unreliable synchronization strategies can be used to deliver expected interactivity without hurting consistency. Future work will be required to meet the synchronization requirements of more complex online virtual environments.

8 Acknowledgements

Thanks to Alex Horn and Alessandro Marianantoni of REMAP for their help testing the sample application, and to Lixia Zhang for feedback on the architecture.

[7]In NDN, the emphasis is on *securing the data* rather than *securing the channel* as currently done in SSL, for example. Cryptographic authentication of signed content objects and encryption of private data are the two main approaches to be explored in the next game project.

References

[1] R.E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA, 1977.

[2] CCNx® Create Collection Protocol. `http://www.ccnx.org/releases/latest/doc/technical/CreateCollectionProtocol.html`.

[3] CCNx® Synchronization Protocol. `http://www.ccnx.org/releases/latest/doc/technical/SynchronizationProtocol.html`.

[4] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, 5(5):440 – 452, September 1979.

[5] E. Cronin, A.R. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools and Applications*, 23(1):7 – 30, May 2004.

[6] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the Internet. In *IEEE Network*, volume 13, pages 6 – 15, August 1999.

[7] S. Ferretti. *Interactivity maintenance for event synchronization in massive multiplayer online games*. PhD thesis, University of Bologna, 2005.

[8] R.M. Fujimoto. Parallel and distributed simulation. In *Proceedings of Simulation Conference*, volume 1, pages 122 – 131, 1999.

[9] V. Jacobson, D.K. Smetters, and J.D. Thornton. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT, pages 1 – 12, New York, NY, USA, December 2009. ACM.

[10] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1 of *INFOCOM*, pages xxxv+2866, March 2004.

[11] Unity^{TM} Car Tutorial. `http://unity3d.com/support/resources/tutorials/car-tutorial`.

[12] Develop^{TM}: The top 10 game engines revealed, June 2009. `http://www.develop-online.net/news/32250/The-top-10-game-engines-revealed`.

[13] S.D. Webb and S. Soh. Cheating in networked computer games: a review. In *Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*, DIMEA, pages 105 – 112, New York, NY, USA, 2007. ACM.