Egal: Making Peer-to-Peer Games over Named Data Network

1st Author Name
Affiliation
Address
e-mail address

2nd Author Name
Affiliation
Address
e-mail address

3rd Author Name
Affiliation
Address
e-mail address

ABSTRACT

Author Keywords

INTRODUCTION

Requirements of massive multiplayer online games (MMOG) from both operators and gamers have been rising. On the one hand, gamers like games that are *interactive*: games should be responsive to user input, and ideally should provide multiple channels for inter-person communication. On the other hand, gamers equally care about *consistency*: all participants should have an equal understanding of the game state at any time, unless it is decided by the game rules. Few customers would be glad if they were told that *interactivity* and *consistency* are two contradicting requirements, though it is indeed the case. To make things worse, requirements from operators are no less complex than those from gamers. First, they want the game system to be *scalable*: service quality should not fall dramatically when the number of gamers is large or is rapidly growing. Second, bandwidth consumption should be minimized for financial reasons. Third, just like any service providers they want a *security* model to survive from attacks. Finally, game operators specially want to *avoid cheating* on behalf of honest gamers.

In [10] the authors proposed *Contnet-Centric Network* (CCN) or *Named Data Network* (NDN), a second generation Internet architecture that reduces network traffic and achieves scalability, security and performance simultaneously (see section NDN Background). NDN treats content as a primitive in stead of location, and retrieves a content by its name instead of its IP address. By making use of web cache, NDN improves content distribution efficiency and reduces traffic. Its one-for-one flow control gives the network scalability and adaptability. Security is embodied in content, which is more trustworthy than securing connections and hosts.

Because NDN has these appealing features, we would like to explore its capability of solving game-related problems. We developed the Egal library which is a C# wrapper of native NDN code and would allow for fast prototyping. Using the library, we adapted an open source car racing game into a peer-to-peer NDN game. Meanwhile, we studied the requirements of consistency, scalability and security in the context of NDN. Our experience would work as an example for future NDN game designers and our library could be reused by future developers.

We include concepts from both game design and NDN in section BACKGROUND. In section NDN GAME DESIGN we present our sample game and its consistency maintenance mechanism. Section IMPLEMENTATION reveals the general approach to developing games on NDN. Then in section DISCUSSIONS we compare NDN games with traditional ones on bandwidth requirement, security and cheating avoidance.

BACKGROUND

Game Design Background

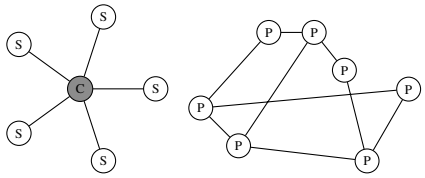
The scalability of a system is closely related to its architecture. In an unscalable architecture, when the client number increases resource bottlenecks would appear, reducing system overall performance. Depending on which architecture is used, a system may need a synchronization mechanism to maintain consistency. Security and cheating avoidance strategies are also architecture-dependent.

Game Architectures

Game architecture are usually classified as Client/Server (C/S) or peer-to-peer (P2P).

In C/S architectures (figure 1(a)) all clients are connected to a central server which receives client events, authenticate, compute and publish updates of global game state. The only two responsibilities of clients are sending user input to the server and rendering the new game state received from it. This centralized model intrinsically guarantees consistency, as there is only one source of global game state. However, the server is also the resource bottleneck, rendering the system unscalable. This architecture is not robust as the server is a single point of failure. Finally the server introduces additional latency.

In contrast, in P2P architectures (figure 1(b)) there is no central authority. Clients, or peers are interconnected with each other and each peer maintains its own copy of the game state. Player authentication and game state computation are decentralized and peers send messages asynchronously to inform updates. Because P2P architectures resource-growing [12], they are scalable. Because peers send messages directly to



(a) C/S (b) P2P

Figure 1. C/S and P2P architectures

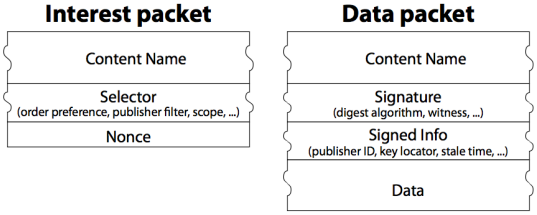


Figure 2. packet types in NDN

each other, network latency is minimized. However, P2P architectures require synchronization mechanisms to guarantee the consistency of the replicated game state among peers. Cheating is also easier in P2P than it is in C/S.

Hybrids of C/S and P2P exist: Mirrored Game Servers, Distributed Scene Graphs and many more. These architectures are described in [11, 12, 8, 6].

Synchronization Mechanisms

The simplest way to maintain consistency among peers is to use totally ordered service, which is often not available. Therefore conservative and optimistic synchronization algorithms have been proposed to prevent or detect (and then correct) misorderings [8].

Conservative synchronization algorithms would allow each peer to process received events only if it is safe to process them. Newly arrived events are usually delayed for an amount of time and the synchronization algorithm can determine the correct execution order. Famous examples are *Chandy-Misra-Bryant algorithm* [5, 4], *Lockstep synchronization* [9] and *fixed time-bucket synchronization*.

Optimistic synchronization algorithms assume received events are in correct order and process them without delay. Then, if late events arrived a *rollback* will be performed and all those optimistically executed events should be canceled and reprocessed. In order to rollback, large memory will used to take multiple snapshots of game state. Also, a mechanism to determine the early bound of rollback (for example the *Global Virtual Time*) will be mandatory. The most famous optimistic synchronization algorithms are: *Time Warp*, *trailing state synchronization* [6] and *optimistic time-bucket synchronization* [7].

NDN Background

NDN *names* can be human-readable (for example `/Egal/Car/Scene0`). Their binary encodings can be used for routing using longest prefix match.

NDN is designed for content distribution. Two packet types exists in NDN: *Interest* and *Data* (see figure 2). A data receiver first initiates an Interest packet that bears the *name* of the *content* it wants. The packet is then routed by name until it reaches a host that has the corresponding content. The matching content object is sent back as a Data packet, satisfying all Interest for it on its way and leaving a trail of 'bread crumbs' [10] in the routers' memory. Later Interests for the same content may be directly satisfied by the routers. It is guaranteed that in NDN every piece of data would flow through a link for at most once. This would reduce a huge amount of network traffic around content distribution centers.

Note that NDN is receiver driven. Flow balance is also controlled by the receiver [13]. Also note that multicasting is intrinsically built in.

CCNx Sync Protocol

The CCNx Sync protocol allows applications to define collections of named data in *repositories* that are to be automatically kept in sync with identically defined collections in neighboring repositories [2]. A collection in the repository is also called a *slice*. A local *Sync Agent* builds and manages a *Sync Tree* for a slice, using names that represent the content of the slice. Hashes of Sync Trees are sent among neighboring Sync Agents to detect any discrepancy. Once detected, normal Interest and Data packets will be sent for inconsistent data.

CCNx Sync protocol should not be confused with the game synchronization protocols mentioned in Synchronization Mechanisms. CCNx Sync maintains data integrity: it keeps local repository slices coherent remote ones. Game synchronization algorithms provide a higher level service: they ensure that sync actions will be taken *in order*.

NDN GAME DESIGN

In this section we present game design issues that are specific to NDN games. In most places we use our game adapted from the Unity3D Car Tutorial [3] as examples but we are not restricted to this genre.

P2P Game Architecture

We decided that our sample game should be P2P because of the following reasons. First, P2P architectures are scalable and robust, with minimized network latency (see Game Architectures). Second, these advantages of P2P can be further enhanced by NDN. NDN reduces traffic in the network, which means that given the same bandwidth, more players could join the game. With data packets cached in the network, data receivers will enjoy a even smaller latency. Third, some of the disadvantages of P2P can be (partly) overcame by NDN. Security, for example, will be enhanced by NDN's content validation and trust management scheme. By naming contents instead of hosts, it will be more difficult for attackers to focus on their target. Several types of cheating behavior can be automatically avoided by NDN (see DISCUSSIONS). Finally, improvements on P2P architectures will also benefit the hybrid architectures. As an example, the Mirrored Game Server architecture has its clients connected to a local server,

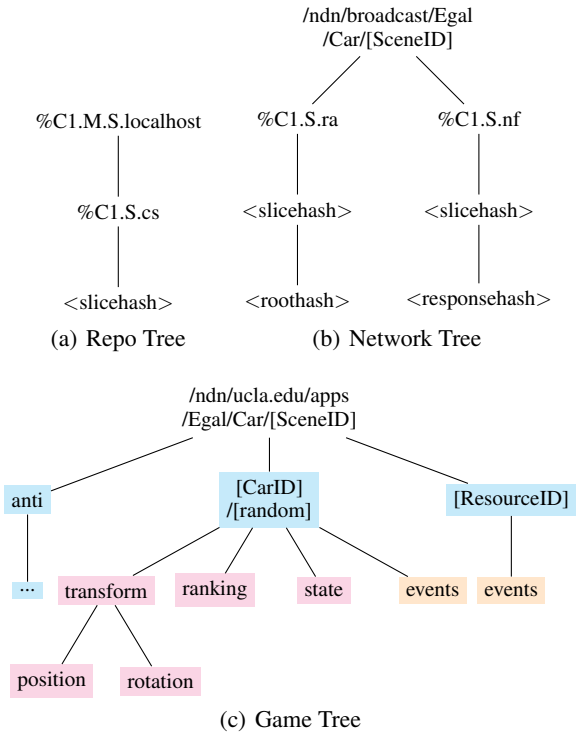


Figure 3. Namespace of our sample car racing game[1]

but typically orchestrates its distributed servers into a high-performance P2P network, so NDN could positively affect it in a similar way.

Namespace Design

Namespace designing is a task specific to NDN application developers. A complete *namespace* of a program is a full collection of names that will be used during the program's execution. Namespace can be presented as trees. For instance, figure 3 is the namespace of our sample car racing game. It is composed of three trees: a *repo tree*, a *network tree* and *game tree*. Each tree node is contains one or more NDN name component(s). One can learn the major functionalities of an application by looking at its namespace design.

Names in the Repo Tree are used for communication with the local repository: we use CCNx Create Collection Protocol to create a slice, which is a collection of content objects with a common name prefix. Two parameters are used to define the slice: *topo* and *prefix*. They later become the root node of the network tree and the game tree. `%C1.M.S.localhost` restricts the scope to the local node; `%C1.S.cs` is a command marker for the Create Slice Interest [1].

The network tree illustrates names used by the CCNx Sync Protocol. This protocol is used for name discovery, which explains why its Interests are broadcasted on the NDN testbed (see its root node `/ndn/broadcast/...`). Note the use of `[SceneID]` in the namespace, which indicates that each scene will have its own slice and be synchronized independently. For small game this may not be necessary; for large games this might be replaced with `[DistrictID]` or `[BlockID]`. `%C1.S.ra` and `%C1.S.nf` are command markers for the Root Advice Interest and the Node Fetch Interest respectively [2].

The game tree is most closely related to the object hierarchy in the virtual world. In fact, a big portion of the tree is constructed by extracting objects and their member variables that needs to be transmitted through network. Please note the use of `[random]` in the namespace. In a car racing game each car should have a unique identifier in the network, therefore we append a `[random]` after `[CarID]` to reduce the probability of name collision. `[random]` can be omitted only if it is guaranteed by the application that `[CarID]` would not collide with one another. The same applies to most user-controlled avatars (first person shooters, heroes played by gamers etc.). However, with global virtual resources (such as a potion, a weapon or a bomb) and program controlled avatars (for example a bot enemy), the opposite is the case. The `[ResourceID]` in the namespace should never be combined with `[random]` and should be named systematically by the application. Otherwise illegal duplications of the resource would plague in the virtual world.

The game tree is designed for optimal transmission and does not have to resemble to the game's object hierarchy. In fact, names for immutable objects and static variables almost never appear in the game tree. The reason for excluding immutable objects is quite obvious: they should be *installed* with the software, not *transmitted* through the network. The exclusion of the terrain object in our game tree is a case in point. As for static variables, they should be *initialized* with their parent object (if there is one) and thus can be omitted in many cases. An example for this is our `[ResourceID]` node. A resource object in our game has many static members (position, value etc.), which are initialized when the resource is dynamically created. Based on this observation, during synchronization we can pack these constant member variables as a content object and name it after the resource's name. The result is that none of these members become a child node of `[ResourceID]` in the game tree. On the other hand, names with little meaning in the virtual world might be added to the game tree just for transmission convenience. The node `state`, for example, is not a member variable of the `car` class, but a *summary* of all variables under `[CarID]/[random]`. Hence, for a peer who is interested in `Car0`'s updates, issuing an Interest for the `state` is more efficient than sending three packets for `position`, `rotation` and `ranking`. [2]

Such name components as `state` can have many variations that serve different purposes. For example, an object may want to keep two summaries of different detail levels: a `short` summary and a `long` summary. The longer summary might

[1]Colors of the nodes will be explained in Asset, State and Event. Nodes surrounded by <> will be generated at run time using algorithms chosen by NDN. Nodes surrounded by [] are variables that will be substituted with multiple values. Note that / are not part of NDN names, they are for illustration purpose only.

[2]Note that the peer cannot just issue an Interest for `car0` and hope to get all the content. An Interest like that will be replied with `car0` or one of its children and the peer would not know which child it is until the data packet is received.

interest players who are directly interacting with the object while the shorter one can be used by distant players for the integrity of their global view. Name components can also be **public** (for everyone including enemies) or **private** (and encrypted, for group members only). An **invisible** name component can be used to provide a summary that does not have position information. In effect, this hides a player from others' view. The interesting fact that name components are not restricted to nouns may inspire NDN developers to design namespaces that are rich in meaning and functionalities.

SYNCHRONIZATION

The purpose of synchronization is to maintain consistent views of the global game state for all players. The definition of global game state varies from author to author, but it generally means the combined state of all entities (players, non-player characters and objects) in the game. In NDN in particular, the global game state is equivalent to the game tree (figure 3(c)) and all its corresponding content objects. Hence, to synchronize a NDN multiplayer on-line game is to synchronize its game tree and all the content objects that underlies.

We divide the problem of synchronizing the game tree into three sub-problems: Asset synchronization, State synchronization and Event synchronization. We recognize nodes in the game tree as *asset*s, *state* and *event*s and use choose different strategies to synchronize them. The idea behind such a division is that assets, state and events have different requirements for synchronization.

Asset, State and Event

Asset, state and event are the three classes of content that need to be synchronized. In figure 3(c) for example, all asset names are colored blue, all state nodes pink and all event nodes yellow. A formal definition is as follows:

Asset A node in the game tree whose presence and absence *cannot* be deduced from the presence and absence of its parent node.

State A node in the game tree whose presence and absence *can* be deduced from the presence and absence of its parent node.

Event A special State that is defined solely for the interaction among the previous two types.

Intuitively, the underlying contents will be called *asset*s, *state* and *event*s too and we will not distinguish a content from its name in many cases. Typical examples of assets are those dynamically created entities that do not have a parent in the virtual world: an avatar controlled by a player, a helmet lying on the ground, a bot controlled by game AI. Their corresponding member variables usually fall into the state class: the health points (HP) of an avatar, the defense power of a helmet, and the position of a bot. Actions taken by assets are usually recognized as events: a "hit" from an avatar to a bot, a "heal" to a group member, a "pick" to a public object, or an "acknowledge as owner" from an object to a hero. Note that although all these examples have some physical meanings in the virtual world, it is not a prerequisite for any of the three types. Assets, state, and events refer to a broader class of content. This can be exemplified by the **anti** asset, which is used to cancel the existence of a character or an object (see Asset Synchronization for details).

Asset, state and event are correlated and they model changes and happenings in the virtual world. An asset can change its own state and publish it, but it cannot change the state of another asset or publish the state for that asset. An asset could, however, publish an event that suggests a state change to another asset. The later asset will "consider" this suggestion and may change its state correspondingly and publish the new state. The following mini story shows this. Alice was a player-controlled warrior in a role-playing game "Killing Zombies". According to our definition Alice is an asset who has a lot of state such as **position**, **HP** and **defensepower**. When Alice was walking around searching for zombies, she was voluntarily updating her **position** state and publishing it to the network so that other players like **Bob** and **Trudy** could see her. When she found a zombie (named zombie0), she published a "hit" event (`.../Alice/events: decrease zombie0's HP`). Zombie0 (and other assets) received this event and found itself killed by Alice. So it published a new state saying that its HP has fallen to zero and is going to die. All players within vicinity will get a visual feedback of Alice's "hit" event (see Event Synchronization for more details). After killing zombie0, Alice found a prize on the ground. It was a helmet which could increase a warrior's defense power by 20 percent. So Alice published a "pick" event (`.../Alice/events: pick helmet#`) and the helmet published a "acknowledge as owner" event in return. The story ends with Alice wearing her new helmet searching for the next zombie. Note that by the end of the story the helmet is not an asset any more since it got a parent object – Alice. Its original name in the game tree will be "canceled" by an anti object and it will become a state (such as **.../Alice/outfit/head**).

Asset Synchronization

Asset synchronization maintains the consistency of all assets. In the game tree, it is in charge of synchronizing all nodes colored blue. By definition, if all asset names are known then the entire game tree became know, as state and events' existence are predictable. Therefore, asset synchronization is also called *name discovery*.

An Unordered Mechanism

Assets' requirement for consistency is relatively low. Updates about assets can be processed in a FIFO order – they do not have to be delayed and sorted by their generation time before being processed. The reason comes three folds. First, assets can only be *created* or *destroyed* during their life time. Second, assets are predefined in a game so they will never change once created. The state of an asset will change, but this will not affect the definition of the asset (its name, structure, attributes etc.). Third, by definition, assets are independent of each other. These features make synchronization of assets very similar to synchronizing a directory of files, without caring about the content of those file content. Files are

also dynamic, independent objects whose content can be regarded as merely "state"s. In file synchronization, the order of which files are added or removed is not important – as long as a directory shows the right files, it is consistent. The same applies for asset synchronization.

Asset synchronization relies heavily on the CCNx Sync protocol. When a peer initializes its game, the local Sync Agent will write one or more slices into its local repository. These slices are units of synchronization and are defined by *topo* and *prefix*. Topo describes the scope within which the sync function should work. Prefix describes the common semantic location of the slice and its collection of data. For example, our topo is `/ndn/broadcast/Egal/Car/[SceneID]` which means the slice is synchronized on the NDN testbed (since its Interests will be broadcasted on it); our prefix is `/ndn/ucla.edu/apps/Egal/Car/[SceneID]` which means that this slice is a collection of assets in a scene of a car racing game, which is developed with the Egal library and is one of UCLA's applications running on the NDN testbed. Once the slice is created, the game application can use prefix to name content objects and write them into the local repository. Sync Agent will guarantee the data integrity of the slice with all other identically defined slices within the scope described by topo.

Managing Assets
Propagation Model and Efficiency

State Synchronization
An Ordered Mechanism with Misses
Bandwidth Consumption
Distribution Efficiency

Event Synchronization
An Ordered Mechanism with No Miss

IMPLEMENTATION

DISCUSSIONS

Bandwidth Consumption

Scalability

Security

Cheating Avoidance

RELATED WORK

CONCLUSIONS

ACKNOWLEDGEMENTS

REFERENCES

1. CCNx Create Collection Protocol. `http://www.ccnx.org/releases/latest/doc/technical/CreateCollectionProtocol.html`.
2. CCNx Synchronization Protocol. `http://www.ccnx.org/releases/latest/doc/technical/SynchronizationProtocol.html`.
3. Unity Car Tutorial. `http://unity3d.com/support/resources/tutorials/car-tutorial`.
4. Bryant, R. E. Simulation of packet communication architecture computer systems. Tech. rep., Cambridge, MA, USA, 1977.
5. Chandy, K., and Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering SE-5*, 5 (Sept. 1979), 440–452.
6. Cronin, E., Kurc, A. R., Filstrup, B., and Jamin, S. An Efficient Synchronization Mechanism for Mirrored Game Architectures. *Multimedia Tools and Applications 23*, 1 (May 2004), 7–30.
7. Diot, C., and Gautier, L. A distributed architecture for multiplayer interactive applications on the internet. *Network, IEEE 13*, 4 (jul/aug 1999), 6 –15.
8. Ferretti, S. *Interactivity maintenance for event synchronization in massive multiplayer online games*. PhD thesis, 2005.
9. Fujimoto, R. Parallel and distributed simulation. In *Simulation Conference Proceedings, 1999 Winter*, vol. 1 (1999), 122 –131 vol.1.
10. Jacobson, V., Smetters, D., and Thornton, J. Networking named content. *Emerging networking* (2009), 1–12.
11. Lake, D., Bowman, M., and Liu, H. Distributed scene graph to enable thousands of interacting users in a virtual environment. In *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games*, NetGames '10, IEEE Press (Piscataway, NJ, USA, 2010), 19:1–19:6.
12. Smed, J. Cheating in networked computer games: a review. *Proceedings of the 2nd international conference on ...* (2007).
13. Zhu, Z., and Jacobson, V. ACT : Audio Conference Tool Over Named Data Networking.