A FRAMEWORK FOR NETWORK-AGNOSTIC MULTIPLAYER GAMES

Patric Kabus
Alejandro P. Buchmann
Databases and Distributed Systems Group
Technische Universität Darmstadt
64289 Darmstadt, Germany
{pkabus,buchmann}@dvs1.informatik.tu-darmstadt.de

Abstract

Producing computer games is a complex and resource-intensive task nowadays. Since this task involves many people with a great variety of expertise, a clear separation of concerns within the project is essential. Especially multiplayer online functionality, which is probably the most popular aspect currently, raises the complexity significantly. Getting this aspect separated allows game developers to focus on design issues rather than on writing sophisticated network code. This paper presents a framework that provides a complete abstraction from network related implementation details.

1. Introduction

“Ten or twenty years ago it was all fun and games. Now it’s blood, sweat, and code.”[4] In the early days, computer games could be developed by a only few people or even a single person. Most of the work was about writing optimized game code for hardware with very limited resources. Due to these resource limitations other aspects of a game, like design, graphics or sound, had to remain very simple. Today’s games are multi-million dollar projects including dozens of highly specialized professionals like 3D artists, level designers, musicians or storytellers. Yet at the core of a game remains a large piece of code, the game engine. It’s purpose is to combine all the digital content (called assets) created by various artists into a playable game.

Since asset creators are mainly artists, they usually have a very limited knowledge about writing code. Thus, the interface to the game engine must require a minimum of programming skills. But at least when creating assets that exhibit behavior (like an enemy which is controlled by the computer) one usually cannot avoid getting in touch with coding. For this purpose, easy-to-learn scripting languages are incorporated into the game engine. Together with predefined methods, which handle common in-game functionality (e.g. a move(x,y) method, which moves a game object to a certain position and automatically performs path finding and collision detection) and can be called from within a script, the programming task is kept as simple as possible.

Hiding complexity gets even more difficult when network gaming comes into play. Network functionality is probably the most important gaming feature today, with networks ranging from a few nodes in a LAN environment to a few thousand nodes in Massively Multiplayer Online Games. Providing a sufficiently consistent view of the game on all nodes of the network is non-trivial. Consequently, asset creators should not be burdened with the task of handling inconsistencies or performing manual synchronization of game objects. However, even programmers will benefit from being shielded from consistency issues. Modern game engines are complex systems composed of various modules. It is generally a good idea to keep consistency-related code within a single module, allowing developers of other modules to focus on their specific tasks. Finally, a clean separation of concerns is a good basis for reusability. Nowadays many game engines are reused by multiple game projects and selling engine licenses is even part of the business model of some producers.

In this paper we present a gaming framework that completely shields game developers from network and consistency issues. Unlike existing game engines, our system does not only abstract from a specific network architecture. Games built using our framework can be deployed in many different environments by simply changing a configuration file. Besides running the game in single player mode locally, we currently support three modes: classic Client/Server, a Peer-to-Peer mode usually known as Replicated Simulation [2] and a hybrid one, called Mutual Checking [13]. In the following we will refer to these modes as CCS, RS and MC respectively. All three modes provide some protection against cheating, an essential property for
today’s games. The underlying abstraction allows developers to extend the framework with their own custom network modes, if necessary. Without the need to commit to a specific network architecture, it is much easier to reuse a game engine in different projects. Furthermore, game developers may allow players of a certain game to change the network architecture by simply altering a configuration file. If a group of players doesn’t trust a single node to host a server for a Client/Server session, they could switch to Peer-to-Peer mode where each node maintains its own local copy of the game state. Finally, home-brew or independent games as well as academic projects may benefit from the possibility of playing around with different network settings without having to change their game code.

In the following section we discuss related work. Section 3 describes the architecture of our framework, while in Section 4 we delve into some implementation details. An example game that we have implemented to show that the framework can actually be used for games is shortly presented in Section 5. Since cheating is an omnipresent issue in online games, Section 6 briefly examines this topic. Finally, Section 7 concludes the paper.

2. Related Work

To our knowledge, no scientific work exists that deals with the complete abstraction from different network architectures within a gaming context.

Kaneda et al. [14] propose a middleware that allows the reuse of Client/Server-based games in a Peer-to-Peer setting. The authors argue that this may be necessary if, for some reason, the producer of a game discontinues to provide the necessary servers. Each player has to install an application on his node which connects to the other player nodes in a P2P fashion. The application acts as a fake server to the local game application by capturing and answering the game related traffic. The global state is synchronized between all nodes, making it appear as if all players were connected to the same server. A major drawback of this approach is that the game’s network protocol must either be openly specified or reverse-engineered. Every implementation of this middleware is specific to a certain game and hardly reusable for other games.

The Real-Time Framework (RTF) [11] also aims at providing an abstraction from the underlying network, but from a different perspective. It does not address pure P2P or hybrid architectures. Instead, it abstracts from the way a multiplayer game is distributed in a multi-server architecture. RTF supports three distribution concepts, namely zoning, instancing, and replication. Similar to our framework, RTF provides a way for game developers to deal with game objects without concerning about synchronization issues. The paper does not go into detail about the underlying network architecture. Thus, it is currently difficult to say in which parts our works complement each other.

Modern commercial game engines usually provide some level of network abstraction, but are mostly tied to a certain network architecture. The technology overview of the upcoming Unreal 3 Engine [10] states that it will be possible to run games either in a C/S or P2P mode. Unfortunately, the architecture is not openly documented and details thus unavailable. It is uncertain whether the engine supports a transition from P2P to C/S or vice versa without altering code. Moreover, it is very unlikely that the engine easily supports hybrid or custom network architectures.

3. Architecture

Our proposed architecture can be divided into three layers and two intermediate interfaces, as shown in Figure 1. The discussion in this section remains on a rather abstract level; important implementation details are addressed in the following section. We start on the highest layer, the game layer, and work our way down to the lowest one.

3.1. Game Layer

This layer contains components like the input manager, the presentation manager and the game engine. The input manager is responsible for accepting commands issued by the player via keyboard, mouse, a gamepad or any other kind of input device. The presentation manager provides the player with an audiovisual representation of the game and probably even some haptic feedback. At the core of any game there is a game engine which manages all the assets that the game is composed of and controls their behavior which is defined by the game logic. The engine may also manage the other components of the game and perform additional tasks like logging in and out of a network game. Although virtually every game is made of components like
those mentioned above, actual implementations may show a
great variety. Professional games today will most likely
consist of much more components, while simple games may
combine everything into a single one. Note that these com-
ponents do not necessarily have to be implemented by the
game developers themselves. There are many implementa-
tions that can be bought off the shelf or are available for
free. The game layer is connected to the lower layers via the
object interface which serves as the top-level abstraction for
our framework.

3.2. Object Interface

The central element of a game is a collection of objects
that constitute the state of the virtual world. The game ob-
jects may represent nearly every aspect of the game: the
players’ avatars, computer-controlled enemies or allies, in-
teractive objects (like vehicles and machines) or completely
static objects (like trees and walls). Even purely logical en-
tities that have no perceptible representation (at least none
that is perceived by a human player), like containers that
aggregate game objects into a logical unit or triggers that
activate in-game actions, may be modelled as game objects.

In a multiplayer game, multiple participants share the
same game world and thus need to have a consistent view
of its state. If the players are located on different nodes of a
network, local copies of the game objects, which as a whole
represent the state, need to be synchronized. Our architec-
ture hides this synchronization effort completely, allowing
a game developer to access and manipulate game objects as
if they were local. Consequently, the interface that is pre-
sented to the developer allows the creation and deletion of
game objects as well as reading and changing their state.
The components of the game that run on a player’s node
may work as usual. E.g., the input manager translates input
events into appropriate changes of the player’s avatar object.
The presentation manager may read the state of the game
objects and generate audio-visual and haptic feedback. And
last not least, the game engine changes game objects when-
ever the rules and the logic of the game require it. Fur-
thermore, the interface provides methods that perform the
necessary bootstrapping when setting up or joining a net-
work session as well as methods that leave or shut down a
session. The following subsection describes the implemen-
tation of the object interface.

3.3. Object Layer

The object layer is responsible for holding up the illusion
that all game objects seem to be local and can be manipu-
lated without concerning about synchronization. Further-
more, it has to handle the necessary bootstrapping when a
new node joins the network or cleanup when a node leaves.

In our framework, every game object has an owner which
keeps a master copy of it. Whenever a node wants to change
a local copy of an existing game object it must send a re-
quest to the owner. If the request is granted, the owner
changes the object state accordingly and sends an update
to every node that keeps a local copy (including the one which
has sent the request). Whenever a node receives an update
sent by the owner of an object, it will perform the contained
change on its local copy. This way we achieve a single-
copy consistency since the owner of an object serializes all
operations on it. Note that in the MC example a group of
region controllers acts as the owner of a game object. Each
region controller in the group receives a request, processes
it independently and sends an update. Whichever node has
a local copy will receive the updates and elect the one which
holds the majority. Please refer to [12] for details, including
a discussion on consistency.

Until now we have only talked about existing game ob-
jects which contain the owner information in their metadata.
What remains is the question of how ownership is deter-
mined when creating a game object. Burdening a game de-
veloper with this task when creating an object would break
our abstraction. To avoid this, the object layer has to pro-
vide a factory method for each supported architecture which
encapsulates the knowledge about determining ownership.
A game developer simply creates an object (using the ob-
ject interface) and, depending on the network configuration,
an appropriate factory is chosen. In our CCS example, the
server is the owner of all game objects and whenever a client
needs to create one, the respective object factory determines
the server as the owner of this object. In contrast, in the RS
example a peer node always takes ownership of objects it
creates. Finally, in the MC setting, the owner id addresses
the whole group of region controllers. As we can see, a
node does not only create objects for itself but it may also
request the creation on another node. Thus, the creation of
a new game object is treated the same way as the manipula-
tion or deletion of an existing one: it is sent as a request
to the future owner. Upon receiving and processing a creation
request, the owner sends an update to all nodes the creation
may concern.

Note that all operations needed for the management of an
object can be mapped onto two types of messages, namely
a request message and an update message. We still need a
third kind of message to inform nodes about organizational
events like the joining and leaving of nodes. Whenever a
node joins the network it sends an announcement to the ex-
isting nodes. Every node that owns a game object which is
relevant for the newly joined node may now send an update
containing the current state of this object. This way, a new
node can be provided with the current state of the game.
When the node leaves again, it may inform the other nodes
that it won’t process request or updates anymore. If the ob-
it is well-established: the Publish/Subscribe method is used to send messages to the appropriate recipients. For instance, a client in the CCS example is never interested in receiving request messages, since it doesn’t own any objects. On the contrary, the server doesn’t care about updates since — due to the fact that it owns all the objects — it is the only one to send them. To complicate matters, nodes join and leave and thus the list of senders and recipients changes dynamically.

However, this problem is not new and a solution for it is well-established: the Publish/Subscribe (pub/sub) paradigm [9]. One of the main advantages of pub/sub systems is the decoupling of message senders from message receivers. Participants of such a system only need to know what kind of messages they want to send. They do not need to know who are actually the recipients of these messages. The other way round, receivers only need to know what kind of messages they are interested in, not who may actually be sending them. The sending of messages of a certain kind is called a publication, while registering interest for a certain kind is called a subscription. The pub/sub system matches every publication to its respective subscriptions and thus takes care that a message will reach its intended recipients. Both, publishers and subscribers, may join and leave dynamically without requiring other participants to take notice of this.

Applying this concept to our framework avoids that owners of game objects and keepers of local copies have to be aware of each other. Any node which wants to manipulate an object simply publishes an appropriate request message. Owners of game objects are subscribed to this kind of message and thus will automatically receive change requests. After processing the request they publish an update and nodes which keep a local copy will receive the change since they are subscribed to update messages. To sum it up, the networking interface has to provide means to issue publications and register subscriptions.

3.5. Networking Layer

The lowest layer of our framework’s architecture is responsible for implementing the pub/sub methods that are offered by the networking interface. Publications have to be routed over the network to the appropriate subscribers. This layer also has to take care of managing publishers and subscribers which dynamically join and leave the network. Please refer to Section 4.2 for a detailed discussion on implementation issues regarding the networking interface and layer.

3.6. Concluding Overview

Figure 2 gives a more detailed overview of our threelayer framework including its two interfaces. On top is the game layer which connects to our framework via the object interface. Within the game layer, one may simply manipulate game objects as if they were local without paying attention to the layers below. The only thing that may be noticeable is a delay until a manipulation actually takes effect. (This delay may be hidden from the player by using commonly known techniques like Dead Reckoning [18].) Below the object interface is the object layer where the configuration of the desired network architecture takes place. A node has to define to which topics it publishes and subscribes and which factory it uses for creating objects with the correct ownership. Supporting different network architectures means providing the appropriate definitions and factories. This layer is also responsible for handling the login and logout of nodes. Finally, the networking interface serves as an abstraction to the message handling. By using a generic interface one may use different implementations in order to fulfill certain performance or scalability requirements or simply to experiment.

4. Implementation Issues

This section will give more insight on some of the implementation issues of the framework architecture. In order to be able to speak of a complete framework, we must provide more than merely a networking middleware. A gaming framework should also provide standard components that are located on the game layer, like the input and presentation managers and the game engine. However, our research focus lies on the network transparency which is not directly related to these components. Additionally, there exists a vast amount of — free and commercial — implementations that may easily be integrated. Consequently, we only provide implementations of these components to the extent they are necessary for our example game (see Section 5). In the following, we focus on the object model and the pub/sub system implementations. The former is the part that game developers have to deal with if they want to create a game that is agnostic of the underlying network architecture. The understanding of the latter is important if one wants to extend the framework with new network architectures or optimize existing ones.
4.1. Object Model

Many different ways exist to manage objects within a virtual gaming environment [3, 5, 7, 8]. We have chosen an approach that provides high flexibility as well as ease of use. It is completely data-driven, i.e., every aspect of a game object can be changed at any time dynamically without the need for a recompilation. This speeds up the development process and should make it easy to integrate this framework into the workflow of a game developer.

The type system of the game object model doesn’t rely on static types defined by the programming language’s class hierarchy (our prototype is implemented in Java). Instead, a generic GameObject class is used which is assigned a game object type dynamically. The type itself consists of a number of state variables and methods plus possible base classes. Every type inherits all of the states and methods from its base types and thus new types can be easily composed of existing ones. Game object methods may be defined in any scripting language which is available for the Java Scripting Platform [16]. The type definition itself is currently written in XML, but by providing an appropriate import plugin, any format may be used. Figure 3 shows how a simple type definition may look like. The example shows a definition of a bomb which contains a two-dimensional position state and a detonation counter. The “tick” method, which is always called by the node owning the object, allows to trigger time-dependent behavior like decrementing the internal counter of the bomb. Note that in our example game, the tick method doesn’t decrease the counter every time it is called but takes into account a variable that contains the amount of time that has passed since the last call. From within the script additional references may be used, e.g., the current game object (“go”), the object manager (allowing access to other game objects) or the object’s state variables.

All game objects are stored hierarchically within a tree. The object interface allows insertion, manipulation and deletion of objects. Furthermore, a query method allows finding objects that match a certain regular expression. Finally, it is possible to define arbitrary groups of objects as views which provide an easy way to access objects that match certain criteria.

Inserting and deleting objects as well as changing their state triggers the notification of registered listeners. A certain listener is responsible for automatically publishing appropriate requests and updates whenever necessary. Listeners may also be used to manage the membership of views. If a new object is created or an existing one changes its state, any listening view may add the object if it matches the view definition. The same way, objects can be removed if they are deleted or do not match the view criteria anymore.

4.2. Publish/Subscribe

To demonstrate how a pub/sub messaging service can be integrated into our framework we have chosen a simple form of pub/sub, a topic-based approach. Later on we
will discuss how more powerful approaches may be used to lower bandwidth consumption or improve scalability.

4.2.1 Topic-based Publish/Subscribe

As the name implies, in a topic-based pub/sub system participants publish and subscribe to topics and each topic represents a certain kind of message. The obvious way to model our communication is to assign each type of message — requests, updates and announcements — its own topic. We first demonstrate how requesting a change and sending an update works within the three example architectures we have implemented. Next, we will show how the announce topic may be used for handling nodes joining and leaving the network.

The following is a short overview of how those architectures distribute the ownership of game objects. For more detailed information please follow the references given in Section 1.

Classic Client/Server (CCS) The central server is the owner of all objects and thus keeps all master copies. Clients only store local copies which are updated by the server.

Replicated Simulation (RS) Each peer may own certain objects for which it keeps the master copies. It stores local copies of the objects owned by other peers.

Mutual Checking (MC) In order to avoid arbitrary manipulations by malicious nodes, each object is owned by multiple region controllers (RCs). Thus, each RC keeps its own master copy of an object and any change request has to be sent to each RC. After changing the state of a master copy, each RC sends an update to the local copies on the clients. The client compares the update messages and elects the one that holds the majority.

Figure 4 shows the request/update process in the CCS context. Client 1 wants to change an object and publishes a message to the request topic. The server which owns all objects has subscribed to this topic and thus receives all requests. After performing the requested changes the server publishes a message containing the changes to the update topic. All clients, including the one that has sent the request, are subscribed to this topic and receive the update.

In the RS context (Figure 5), a peer that wants to change an object publishes a request. All peers within the system are subscribed to the request topic, but only the owner of that object needs to process the request. The state update is then published and received by all peers, since each of them is subscribed to the update topic.

Figure 4. Request/update in CCS mode

Figure 5. Request/update in RS mode

Our last example, the MC context (Figure 6), is very similar to the CCS setting. Instead of having a single server, all RCs are subscribed to the request topic. After performing the requested change, each RC publishes an update. The clients, which are subscribed to the update topic, receive all updates from the RCs. Before an update will be performed, the correct one is elected out of the received updates.

To handle events like nodes logging in and out of the system, a third topic, called announce, is used.

Whenever a new player joins the game, an object has to be created that represents that player. The nodes already in the system need to be informed about the state of this new player object. Figure 7 illustrates this process in the CCS context. The server, which is subscribed to the announce topic, receives a login announcement published by the new client. It creates a new avatar object representing that player and publishes an appropriate update. This update is received by all clients, since they are subscribed to the update topic.

After logging in, the new client needs to be supplied with the current state of the game. For this purpose, every node that owns game objects must be subscribed to the announce topic. Upon receiving the login message, the owners may publish an update containing the complete state of their
master copies. Unfortunately, publishing the whole state of all master copies every time a node joins the game would be a waste of bandwidth. Every node subscribed to the update topic would receive the current state, even if its local copy is up-to-date. Optimizations that avoid this are discussed in the following subsection.

If a node wants to leave the network it simply publishes a log-out announcement. After receiving this message, the server publishes an update that removes the avatar object of the corresponding player from the game. Note that in our RS setting things are slightly more complex, since a leaving peer node may be itself the owner of certain game objects which are still needed. Before leaving the network, the node has to make sure that these objects are transferred to other peers. In order to do so, it can request the creation of an object on another peer by specifying this peer’s id as the owner id.

### 4.2.2 Optimizations

An important way to reduce network bandwidth requirements in online games is to restrict the amount of updates a certain node receives. Obviously, a node does not need to be informed about changes of game objects that the local player can neither perceive nor interact with in any way. Limiting the update message to ones relevant for the player is commonly known as Interest Management. Instead of subscribing to all messages that are published to the update topic, a filtering based on the in-game position of objects may be performed.

For example, the Java Message Service [20] combines a topic-based pub/sub approach with filtering based on key/value pairs. Every update published may be enriched with additional properties that contain the position of the updated object. Only when the player’s avatar is in the interaction range of that object the update will be sent to that player’s node.

Instead of using a flat topic space, a hierarchical one may be employed to restrict messages to certain game regions. This approach is usually referred to as subject-based filtering [17]. E.g. in a game that uses a real-world setting, subjects like Earth, Earth.Europe and Earth.Europe.Germany could exist. Whenever an avatar enters a region (e.g. Germany) the node subscribes to the corresponding subjects. On the one hand, this makes sure that the node won’t be bothered with unrelated messages of events that happen in a different country or even on a different continent. On the other hand, the node will receive messages of events that are relevant for the whole continent or even globally. Naturally, changes made by the node will be published to the appropriate subjects in the same manner, depending on their relevance.

Not only the addressing model but also the implementation of a specific model has an impact on performance and scalability. One very important performance criteria of network games is the latency when propagating updates of game objects. Usually nodes of gaming networks talk directly to each other, be it a client talking to a server or peers talking to each other. The delay of changing an object (i.e. issuing a request and getting a reply) equals the roundtrip time between nodes. In an implementation that wants to avoid higher latencies, a node that requests the change of an object must send the request directly to the owner node. Afterwards, the owner has to send its updates directly to all nodes which keep a local copy of the updated object. This way, extra delay caused by additional hops on the network path is avoided. In such an implementation a local software component running on each node can provide the pub/sub interface to the object layer. Internally, this component stores for all topics it publishes messages to a list of all subscriber nodes. Whenever a node publishes a message it can send it directly to the appropriate nodes. The sub-
scription management service may be located on a separate node. Every time a node subscribes for a topic, the management service can inform the publishers about it. By sending a so called advertisement, a node can inform the management service about its intention to act as a publisher for a certain topic.

A further optimization is that whenever a node wants to change a game object that it owns, it may directly publish an update without the need to send a request first. But one should be aware that this may affect fairness. While the change is propagated to other nodes with the delay of a single hop it is perceived nearly instantly on the local node. This may enable the local player to react much faster than players on remote nodes. To avoid this, an artificial delay may be introduced (e.g. Local Lag [15]).

While the implementation above minimizes latency caused by network delays, it severely limits scalability. Think of a node in a Replicated Simulation which has to send updates to a very large amount of other nodes in the game. This way a node will soon reach the limits of its network connection, especially when using an asynchronous DSL connection with a very limited upload bandwidth. This is where pub/sub systems that rely on intermediate brokers play out their strength. While introducing additional delays for message delivery, the intelligent routing and filtering mechanisms can minimize bandwidth and connectivity requirements on the game nodes.

5. Example Game

For demonstrating the feasibility of our approach, we implemented a game that includes many important aspects found in today’s games. These aspects include a graphical representation, changes in object state through player input or progress of time and interaction between game objects. While in our example they remain very basic, our framework imposes no limits onto their implementation. Rich three-dimensional graphics and sound are possible as well as control of game objects through complex artificial intelligence.

Our game is a simplified version of a famous multiplayer game concept that has been implemented by the open-source game XBlast [1]. Every player controls an avatar which may move freely around the game field. By pressing a button, he can place a bomb at his current location. Placing the bomb starts a timed detonator and when the countdown reaches zero the bomb explodes. All avatars that are in the vicinity of the detonation are removed from the field and, as in the original XBlast game, the last remaining player wins. Figure 8 shows a screenshot of the game.

As intended, the same game code can be used within all three network architectures without any modifications.

6. A Word on Cheating

No multiplayer online game today can come along without some protection against cheating, since the possibility to cheat poses a major threat to the fairness of the game.[6, 12, 19] Fairness is a critical factor for enjoying a game and consequently cheating may drive away paying customers. However, we will not delve into that topic. Instead, we only want to point out that the level of cheat-resistance is determined by the implemented architecture, not by our framework. In the classic C/S setting, all trust is imposed on the server and our framework doesn’t change this. A P2P node within the Replicated Simulation is responsible for the object it owns. However, all peers receive updates about changes of that object and they may check themselves if those changes conform to the rules of the game. Otherwise they may reject an update. In the Mutual Checking scenario, each RC votes for a certain update. The larger the group of RCs is, the less likely it is for cheating nodes to insert a falsified update.

The only thing the framework has to guarantee is that no one is able to forge messages. E.g., if a node receives an update, it must be sure that the sender is really the owner of that object. Nodes may simply identified by IP addresses or, if a higher level of security is necessary or object ownership must outlast network sessions, cryptographic signatures may be used. For this purpose a public key infrastructure is necessary which can be run by the game provider.

7. Conclusion

In this paper we have presented a framework that provides a game developer with a complete abstraction from network related issues. The framework can be divided into three layers: on the highest level the game layer, underneath the object layer and at the bottom the networking layer.
On the game layer, standard components, like the game engine and components managing audiovisual feedback and player input, are located. This is also where a game developer has to implement the rules and the logic of a specific game. All components on this layer communicate through an interface to the layer below, the object layer. Game developers can create, manipulate and delete all game objects as if they were local; network consistency as well as ownership management is handled automatically. The networking interface below hides network related issues behind a publish/subscribe abstraction. If it is necessary to optimize the network layer for different quality requirements, like higher scalability or lower latency, custom implementations can be used.

With network implementation details hidden, game developers can focus more on game design rather than writing specialized code. Implementation details like data-driven game objects further emphasize this approach.

References

[1] XBlast. xblast-center.com