
Egosoft

**Network Integration
Design Plan
For X4**

Version 0.5D

Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

Revision History

Date	Version	Description	Author
05/02/08	0.5	Initial draft	Stefan Hett
05/02/08	0.5D	Adjusted version for the diploma thesis.	Stefan Hett

Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

Table of Contents

- 1 Introduction.....3
 - 1.1 Purpose.....3
 - 1.2 Scope.....3
 - 1.3 Definitions, Acronyms, and Abbreviations.....3
 - 1.4 References.....4
 - 1.5 Overview.....4
- 2 Network Layer Design.....5
 - 2.1 XU Layer (Layer 3).....6
 - 2.2 Network Base Layer (Layer 2).....6
 - 2.3 Framework Layer (Layer 1).....6
- 3 Network Engine Integration.....7
 - 3.1 Iteration 1: Engine Refactoring.....7
 - 3.2 Iteration 2: Replication Support.....11
 - 3.3 Iteration 3: Introduction of Object Roles.....12
 - 3.4 Iteration 4: Setting receivers for objects.....13
 - 3.5 Iteration 5: Object Synchronization.....13

Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

1 Introduction

1.1 *Purpose*

The purpose of this document is to keep a record of the changes made to X4's game engine and to summarize the integration of the network engine. It describes design decisions made and provides the reasons which stand behind these decisions.

1.2 *Scope*

It's outside the scope of this document to describe all the code changes in too much detail. If you need to know the detailed code implementation, refer to the commented sourcecode directly.

Furthermore this documentation will not cover any side-by-side changes made to the game engine which is not directly associated with the integration of the network engine.

1.3 *Definitions, Acronyms, and Abbreviations*

Any new acronyms are covered in appendix A. Furthermore the “Network Integration – Background document” (see chapter 1.4) provides more definitions, acronyms and abbreviations used in this document.

Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

1.4 References

Title	Report Number	Date	Publisher
X4 – Networking Background Information	Revision 1.4D	04/02/2008	Egosoft
doc\diplomathesis\X4 - Network Integration - Background 1.4D.pdf			
X4 – Network Integration - SRS	Revision 0.95D	04/02/2008	Egosoft
doc\diplomathesis\X4 - Network Integration - SRS 0.95D.doc			
X4 – Network Integration - Analysis	Revision 0.9D	04/02/2008	Egosoft
doc\diplomathesis\X4 - Network Integration - Analysis 0.9D.pdf			

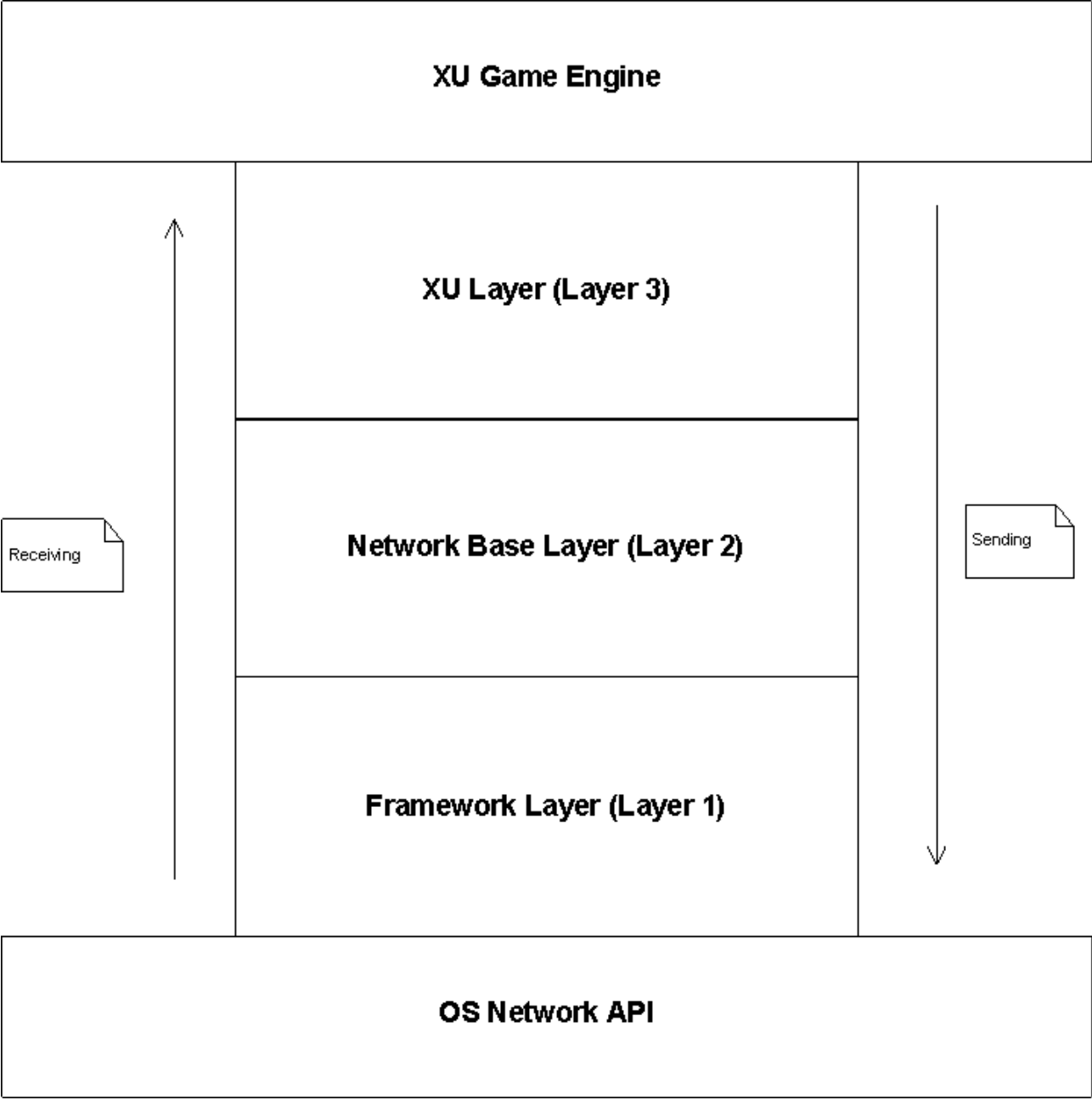
1.5 Overview

Chapter two provides a furrow overview of the network engine's layer structure, while chapter 3 describes all the stages of the network engine's integration process.

Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

2 Network Layer Design

This section provides a summarized overview of the network engine's layer structure. It is essential to understand this structure in order to be able to follow the implementation stages. A more detailed explanation of the network engine's different layers can be found in chapter 2.5 of the “X4 – Network Integration – Background” document. The design described there depicts also the state of the network engine as it was right before implementing the first iteration (see chapter 3.1).



Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

2.1 XU Layer (Layer 3)

The XU Layer contains all classes which can be considered part of the game engine and are directly related to the network base layer (i.e. use objects of layer 2 classes or inherit from layer 2 classes).

There are some exceptions to this caused by the current layer separation procedure not being completely implemented, yet. These exceptions are that classes derived from NetworkObject (a layer 2 class) won't belong in this layer, but remain part of the game engine instead. This is due to a missing NetworkObject interface class which will be added in the near future.

The same applies to classes relying on the Replicable interface, which currently resides in layer 2 but is slated to be moved out of the network engine.

2.2 Network Base Layer (Layer 2)

This is the separation layer which ensures platform interdependency of the network engine. It provides all the functionality which is needed by the XU Layer to provide the multiplayer capability of X4.

By acting as an interim layer between the XU and the Framework Layer, this prevents the XU Layer and any other part of the game engine from having to be adjusted, if the Framework Layer is replaced.

2.3 Framework Layer (Layer 1)

The integration of the network framework is what this layer is used for. The Framework Layer provides those functionality needed by the Network Base Layer to establish the communication to remote hosts, receive data and provide basic networking features.

Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

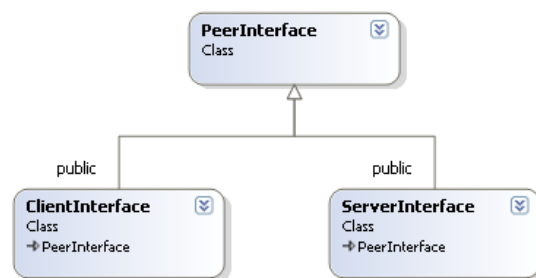
3 Network Engine Integration

This section covers each iteration of the network engine's integration process.

3.1 Iteration 1: Engine Refactoring

The initial tests left us with a somewhat clumsy integration of a rudimentary network engine providing only some basic functionality. To clean things up without throwing everything away, the current implementation will be refactored.

To clarify the communication paths between the network layers, network layer 2 (Network Base Layer) and 3 (XU Layer) will be added three interfaces, each. These interfaces specify the methods which are expected to be provided by the underlying layer.



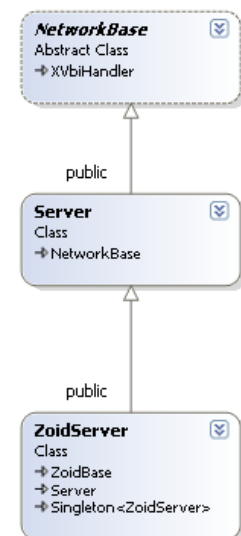
The PeerInterface class (later renamed to HostInterface) defines those methods which are to be implemented by both classes (server and client), while the server- and client interfaces specify additional required methods.

Communication between layers is only possible using the methods declared in these interfaces. This helps to reduce the impact on any higher layer, if we need to replace the network framework in use.

To disentangle the different layers even further, we need to revise the inheritance model which suffers the following design flaw:

One of the intended requirements is that any change made to the Framework Layer should at most result in necessary changes to the Network Base Layer but explicitly not to the XU Layer. However, when we construct a server- or client-object in the game engine, we need to directly create a ZoidServer or ZoidClient object (which belong to layer 1), because of the current inheritance structure.

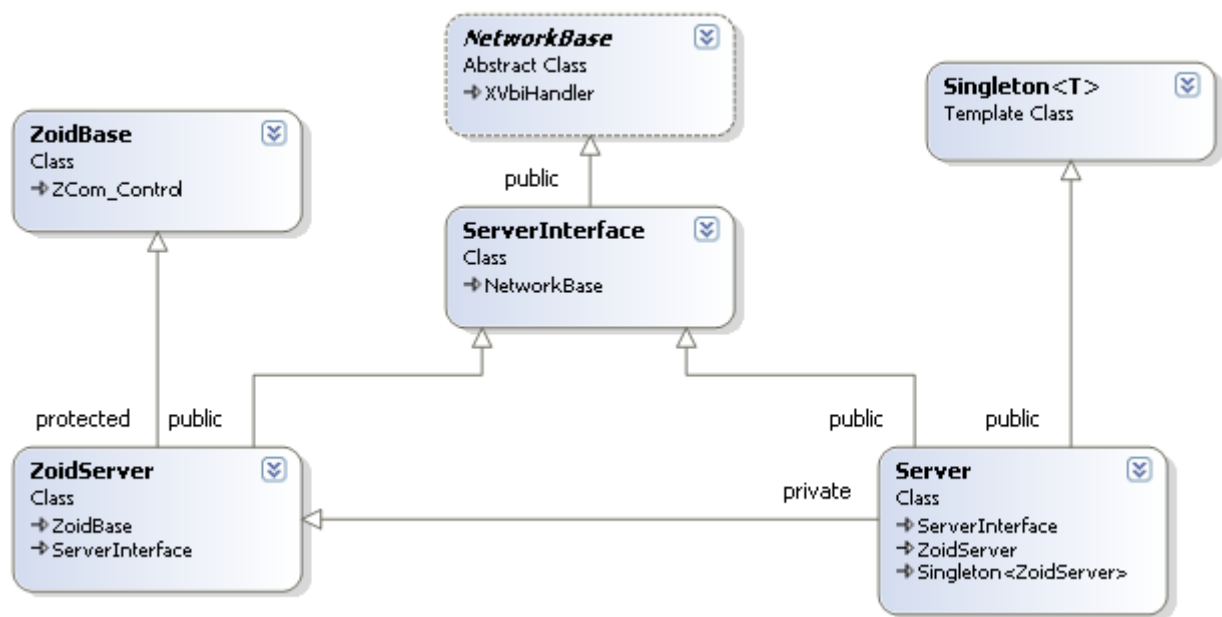
This violates the requirement discussed above.



Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

What we'd like to do is creating a Layer-2-Server-object rather than a Layer-1-Server one. That way we won't have to change the construction of this object in the game engine, even if we completely replace the Framework Layer (due to renamed classes in that layer).

In order to achieve this we'll redesign the inheritance structure and only allow inheritances to go downwards through the different layers rather than upwards. The following class diagram shows the new design:



The first change will be to turn around the inheritance between the ZoidServer- and the Server-class. This also leads to a changed order of calling virtual methods and moves the responsibility of calling those from the Framework Layer to the Network Base Layer.

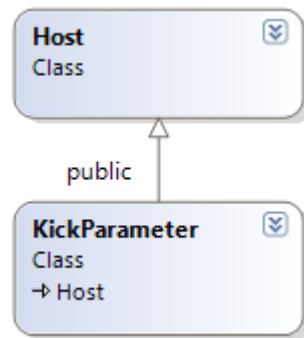
The old style was to call the Server's Init()-method in ZoidServer's Init()-method. Now it's exactly the other way around (Server::Init() calls ZoidServer::Init()).

The last noteworthy change is that ZoidServer no longer inherits from Singleton. Instead Server will inherit from the Singleton class. This allows us to call Server::GetInstance() from anywhere in our game and always get the correct server class, even if we replace the Framework Layer (i.e. ZoidServer with RakServer, for instance).

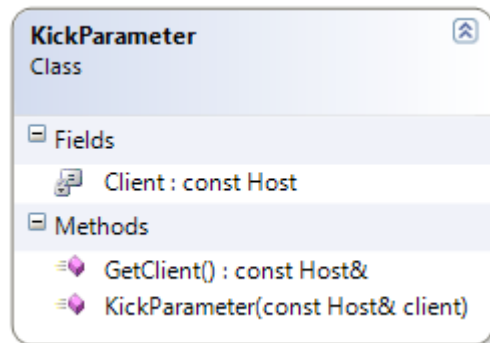
Corresponding changes will be made to the Client class inheritance structure.

Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

To complete the refactoring of the network layers the usage of our own “datatype-classes” like the Host-class will be changed, too. The old way of using these classes was to inherit from these. For instance KickParameter inherited from Host:

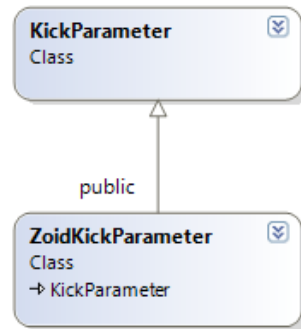


Since this is not in accordance with the methodology of object oriented programming (i.e. KickParameter is not a Host, as the inheritance would suspect), this will be changed by replacing the incorrect inheritances by storing the “datatype” as a member variable instead. In case of the KickParameter class this would result in the following change:



Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

Finally the direct dependency between layer 1 and layer 2 will be eliminated by removing the inheritances between layer 1 and layer 2 datatype-classes.



Given the example above, ZoidKickParameter will be removed and the layer 1 method which used it (i.e. ZoidServer::KickClient()) will be refactored to accept a KickParameter object instead.

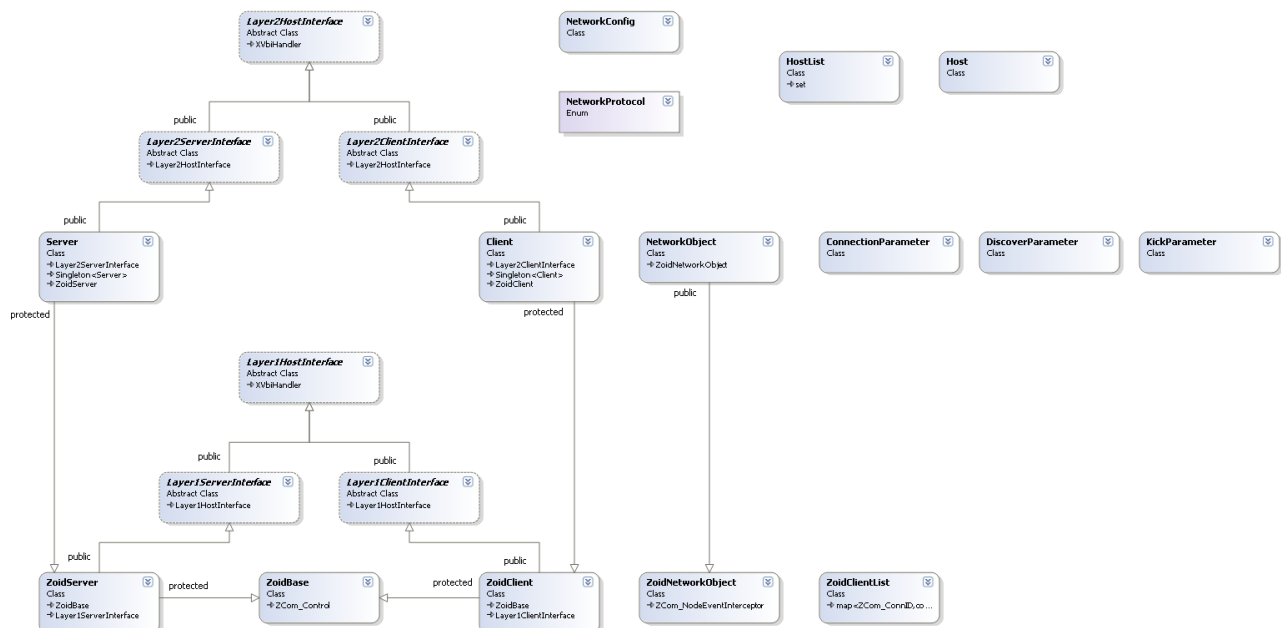
That way a layer only needs to know two things about it's superior layer:

- the interfaces it needs to provide
- the definition of the datatypes used as parameters for some of the interface methods

This leads to Layer 1 no longer having to be aware of the HostList-system which belongs to Layer 2.

Additionally this enables us to get rid of the ServerList/ClientList classes as well as the ServerInfo/ClientInfo ones, which are not needed in Layer 2 anymore.

The following figure shows the class structure as it looks like after the refactoring process:



Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

3.2 Iteration 2: Replication Support

To get things started (i.e. the network engine to synchronize the universe), we'll have to add support for object replication. This will allow us to inform hosts of newly created objects and is necessary for the initial synchronization.

First thing to do (relating to the Network Integration Analysis) is to add a new BitStream class, which will contain the parameters for the static factory method which is to be added to the Component class and potentially other none-component classes. Since ZoidCom and RakNet already provide BitStream classes, we don't need to implement the basic functionality again in layer 2. Instead we'll define a new interface which specifies the methods which need to be implemented by any BitStream class. This interface class is then to be implemented by a wrapper-class in layer 1.

Furthermore we need a way to communicate the replication procedure between layer 1 and layer 2.

To accomplish this, we should inform layer 1 of all the objects which can be replicated. This is necessary for those frameworks (like ZoidCom) which have their own replication system in place and use their own generated class ids to identify the class which needs to be replicated. On layer 2 we use our own ids which potentially differ from those used in layer 1. Therefore we need layer 1 to keep an association list between its class IDs and those we use in layer 2.

This will be done by adding the RegisterReplicationClass()-method to the Layer1Host interface. This method will be called for each replicatable class and receives its class ID.

RegisterReplicationClass() will be called from within RegisterReplicationClasses(), another new method, which resides in Layer2Host and will be called during the server/client initialization.

Since the implementation of these two methods are likely to be the same for server and client classes, it is obvious that the implementation should only be done once. In layer 1 this will be done in the ZoidBase class, in layer 2 a new NetworkBase class will be introduced and implement the functionality there. However this requires a small change in our inheritance structure, since ZoidBase and NetworkBase need to be inherited from LayerXHostInterface, which contains the pure virtual declaration. To work around the diamond problem, which would be caused by the changed inheritance, we use virtual inheritance wherever a class is derived from Layer1HostInterface.

That concludes the necessary steps for registering replicatable classes.

Though there are still two more things to do, to come up with a fully functional replication system:

In order to get a NetworkObject's class id, we add the GetClassID() method to the base class and overwrite it in the derived classes to return a unique ID.

The problem with this generated ID is that it can only be assumed that the returned value is unique for all those classes which are derived from the specialized network object class. It's not unique on a global basis.

To ensure the uniqueness for all class ids, we add a prefix before the returned class id. This prefix is unique for all classes directly derived from NetworkObject and is returned by the GetPrefix() method which is expected to be implemented by all classes which derive from NetworkObject.

Network Integration	Version: 0.5D
Design Plan	Date: 05/02/08
X4-NET-PLAN	

3.4 Iteration 4: Setting receivers for objects

Not covered in the analysis document is a way which allows us to set the recipient for objects. However such a functionality is necessary, if we want to have the control over which objects are being sent to which clients.

The initial idea of adding it in the network base layer was rejected due to the fact that ZoidCom already provides a more or less easy way to do it. Based on ZoidCom's ZoidLevels, an implementation is quite easy.

The integration will consist of two new methods which are being added to the Layer1NetworkObject interface:

```
bool AddReceipient(const Host& receipient);
bool RemoveReceipient(const Host& receipient);
```

3.5 Iteration 5: Object Synchronization

First thing to implement is the Replicator class. This class will allow us to implement the final replication system.

Second thing to do is to add two new methods to the NetworkObject:

```
ReplicationSetup()
InitialReplicationSetup()
```

Both methods will populate a vector with replicators.

The vector populated by ReplicationSetup() is used to set up object synchronization as well as for the savegame system.

InitialReplicationSetup() will fill up a vector, which is being used to create or reconstruct the bitstream used for object replication. Note that this method has no effect on the savegame system.

Both methods will be called from within InitNetworkObject().

At first InitialReplicationSetup() is executed. If the object is constructed due to a replication request from the server, the vector is used to deserialize the received bitstream.

If the object is constructed as an owner object, the replicators are used to create the announcement datastream which is to be send along with any replication request.

Bare in mind that the announcement datastream is constant; that said, it will only be set during the first call to InitNetworkObject() and can't be altered lateron.

After InitialReplicationSetup() has been called, ReplicationSetup() is being executed, the replication setup passed along to layer 1 and the relevant replicators for the savegame system are being stored in a member variable.