
Egosoft

**Network Integration
Design Analysis
For X4**

Version 0.9D

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

Revision History

Date	Version	Description	Author
04/12/07	0.5	Initial Draft.	Stefan Hett
07/12/07	0.6	Clarified some statements in 3.1 and 3.2 and added chapter 3.3.	Stefan Hett
11/12/07	0.7	Revised section 3 (completely changed all chapters before the RPC one)	Stefan Hett
04/02/08	0.9	Completed chapter 1 and partly chapter 2.12. Moved chapter 2 “Different Approaches” to Appendix C and revised it Moved chapter 5 “References” to Appendix B Removed the idea of a proxy list from chapter 3.1. Completely revised the entire document.	Stefan Hett
04/02/08	0.9D	Adjusted version for the diploma thesis.	Stefan Hett

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

Table of Contents

1 Introduction.....	3
1.1 Purpose.....	3
1.2 Scope.....	3
1.3 Definitions, Acronyms, and Abbreviations.....	3
1.4 References.....	3
1.5 Overview.....	4
2 Necessary game engine adjustments.....	4
2.1 Object Roles.....	4
2.2 Object Replication.....	5
2.3 Object Synchronization – Export()/Import() Refactoring.....	6
2.4 Object References.....	8
2.5 Remote Procedure Calls.....	11
2.6 Event System.....	13
2.7 AI.....	13
2.8 UI.....	13
2.9 Movement Controller.....	13
2.10 Mission Director.....	13
2.11 Network Architecture.....	14
2.12 Object Handover.....	14
2.13 Separate network thread.....	14
Appendix A - Glossary.....	15
Appendix B - References.....	16
Appendix C - Different Approaches.....	17
Object Replication and Synchronization Approach.....	17
Synchronized Game Graph Approach.....	18

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

1 Introduction

1.1 *Purpose*

The purpose of this document is to analyze the necessary game engine adjustments in more detail and come up with a way to do the fundamental implementation steps. It is meant to provide the information necessary to do the complete integration and prevents us from running into any pitfalls later during the coding.

1.2 *Scope*

Though this document covers all main problems, it is out of the scope of this document to solve all detailed implementation issues or to come up with a detailed implementation plan (which is the purpose of the following document the “Network Integration Plan”).

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.

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			

Further external references are covered in appendix B.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

1.5 Overview

Section 2 covers the fundamental game engine adjustments .

In addition to this chapter appendix A provides the definition of used abbreviations while appendix B contains a list of further references. For the curious reader appendix C explains the underlying ideas which led to the proposed network integration procedures.

2 Necessary game engine adjustments

It's obvious that neither of the initial approaches (see appendix C) covers all points specified in the X4 - Network Integration – SRS 0.95D document.

Therefore the following chapters list those things which need to be taken care of.

2.1 Object Roles

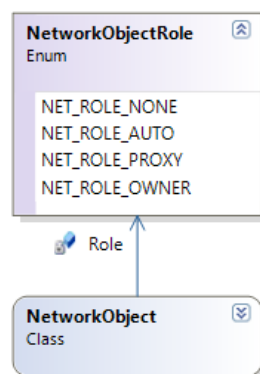
Game objects can be split up in three classes:

- game objects which need to be synchronized between peers
- those which don't need to be
- those which must not be

This has to be taken into account before any data is being sent rather than blindly transmitting everything and then sort out on the client side which received information need to be processed and which can be ignored, in order not to stress the limited network bandwidth with transmitting unnecessary data.

Therefore we need the ability to flag our game objects for network transmission.

The suggested solution is to derive all objects which potentially need to be synchronized over the net from the NetworkObject class.



By setting the object's Role variable, one controls its synchronization behavior. When set to

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

NET_ROLE_NONE, the object will not be synchronized at all. When set to NET_ROLE_OWNER, the object will be synchronized with all peers for which the corresponding object is set to NET_ROLE_PROXY. On the other side NET_ROLE_PROXY objects can synchronize the corresponding NET_ROLE_OWNER object.

The Role value will default to NET_ROLE_NONE. That way we ensure that developers who are currently working with code which is not network related in their case but still relies on classes which are used for network synchronization in other parts of the game engine, don't have to take care about anything.

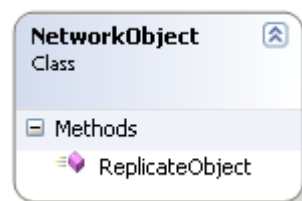
NET_ROLE_AUTO can be used instead of directly setting objects to NET_ROLE_PROXY or NET_ROLE_OWNER which makes working with the network engine in X4 a bit easier. When an object is created and being set to "auto" its Role variable will default to NET_ROLE_OWNER. On the other hand, when its being created due to object replication, it "auto" will correspond to NET_ROLE_PROXY.

2.2 Object Replication

The straight forward implementation for replicating objects over the network is to call the default constructor of the class which needs to be created, once a client receives a replication message from the server. The obvious limitation in this case is that we can't pass along any parameters with the constructor call. To work around this limitation, we could provide the possibility to add an optional bitstream which would be passed along to the constructor. This will work in all cases as long as each synchronized object is able to construct itself.

However that's not the case for components in the game graph, which need to be constructed using the factory pattern due to the requirements of the component system (macros, savegames, patches, etc.).

That's why we use another way to do it. Instead of directly calling the class' constructor, we require each class derived from NetworkObject to provide a class method which will process the replication request.



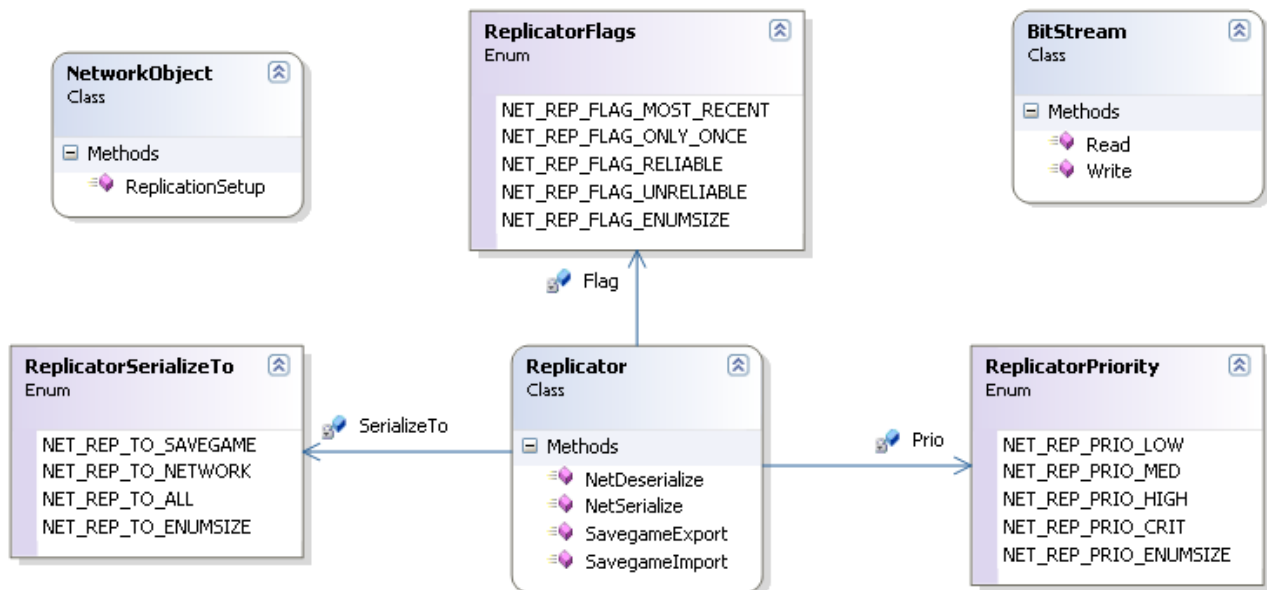
The method will receive a bitstream which optionally keeps any encoded parameters and is expected to return a pointer to the newly constructed object.

Object replication is directly related to the object role. Only objects which role are set to NET_ROLE_OWNER will be replicated.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

2.3 Object Synchronization – Export()/Import() Refactoring

Object synchronization relies on replicators close to the description given by the ZoidCom documentation and is implemented in the following way:



A replicator provides methods used to serialize the data it keeps/represents for both; the savegames and the network transmissions. While the `Net(De-)Serialize()`-methods are supposed to work with size optimized bitstreams, the `SavegameExport()/-Import()`-methods will construct an XML-formatted representation of the replicator.

That way allows us to kill two birds with one stone. Instead of having to code `Export()/Import()`-methods for the savegame system and setup replicators for network transmissions, we combine both.

Replicator classes provide three different ways to properly set up their behavior. The “`SerializeTo`” member variable indicates whether the replicated data will be synchronized over the network, written to a savegame or both while the “`Prio`” variable can be used to set up the replicator's priority used when sending data over the net. Finally the “`Flag`” variable sets the behavior of transmitting the data.

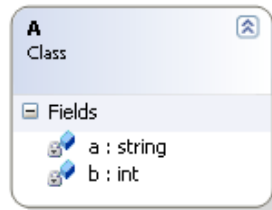
Note that the diagram shown above is not complete. For instance the `NetworkObject` needs either a `SavegameExport()/-Import()` method, too, or some setup data which defines how the complete class is serialized to/from a savegame. Furthermore the `Replicator` will likely need additional member variables used to set up its correct behavior when writing/reading data to/from an XML-node (for instance the name of the attribute the variable is saved to).

Default replicators are provided for all basic data types and a couple of widely spread classes (for instance strings).

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

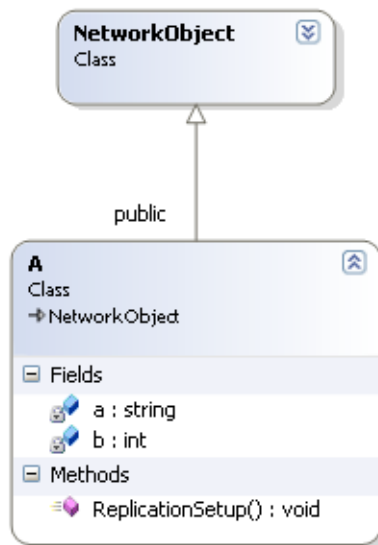
Though the developer no longer needs to write any Export()-/Import()- methods he will need to set up the replicators properly by overwriting the NetworkObject's ReplicationSetup()-method.

The following provides an example of how the new object replication might look like. Imagine you want to synchronize the following class:



a should solely be saved to the savegame, while b should be synchronized over the network as well as saved to savegame.

First thing to do is to derive A from NetworkObject and overwrite the ReplicatorSetup()-method.



The implemented ReplicationSetup()-method could look like this:

```

void A::ReplicationSetup()
{
    new StringReplicator(&a, "name", NET_REP_TO_SAVEGAME);
    new IntReplicator(&b, "power", NET_REP_TO_ALL, NET_REP_PRIO_CRIT,
NET_REP_FLAG_UNRELIABLE);
}
  
```

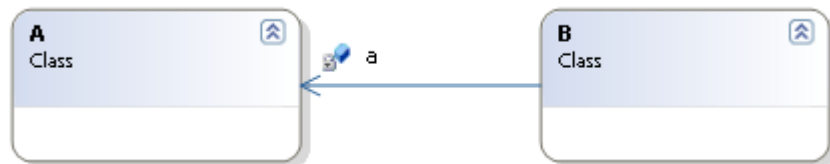
The given example does not claim to be complete in regards to the constructors parameter lists.

Also keep be aware that object synchronization is directly related to object replication, meaning that only objects which have been replicated using the object replication system will be synchronized over the network.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

2.4 Object References

References between objects are handled separately because they might implicitly define object dependencies. The following example illustrated this:



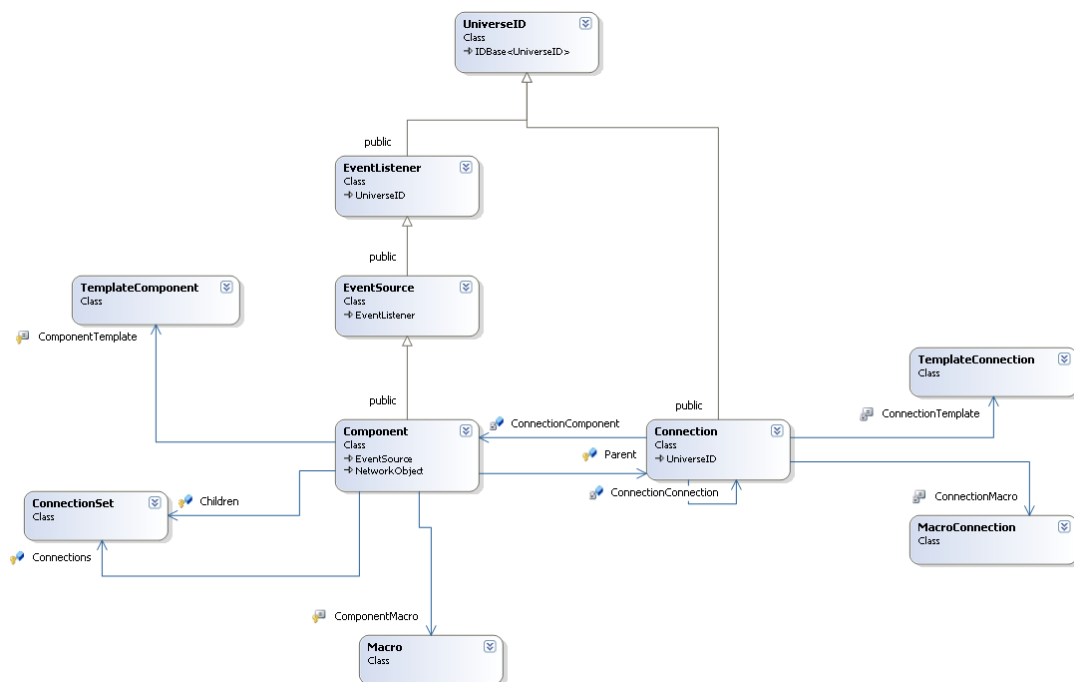
Object B keeps a pointer to object A. Furthermore we assume that object B depends on object A (since it's likely that any member method in B accesses A).

This provides a couple of challenges when synchronizing these classes for us.

1. We can't simply send the pointer directly to a peer (since the address stored in the pointer is most likely pointing to something completely different on the other computer).
2. When object B is flagged for synchronization, object A must be implicitly flagged along with it, because of the dependency.
3. We must be aware of potential cross references (i.e. A could also keep a pointer to B in a different scenario).

There are three ways we keep references within our game engine at the moment:

1. Our game graph keeps references in the way of connections (parents, children, side-by-side connections). The following graph shows the relevant classes used by this system:

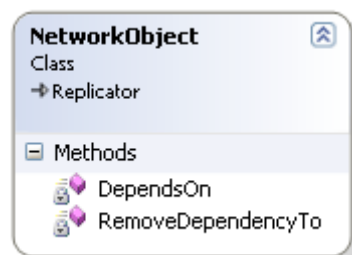


Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

2. References regarding non-components are not stored in the game graph but only as member references, pointers or IDs in objects.
3. Indirect dependencies are not stored in any form, but implicitly assumed.

A suitable solution comes in form of a two stage implementation:

The first stage is a dependency system which creates a “dependency-graph” used to identify objects that need to be synchronized along with others. The developer is provided with two new methods in the NetworkObject class:

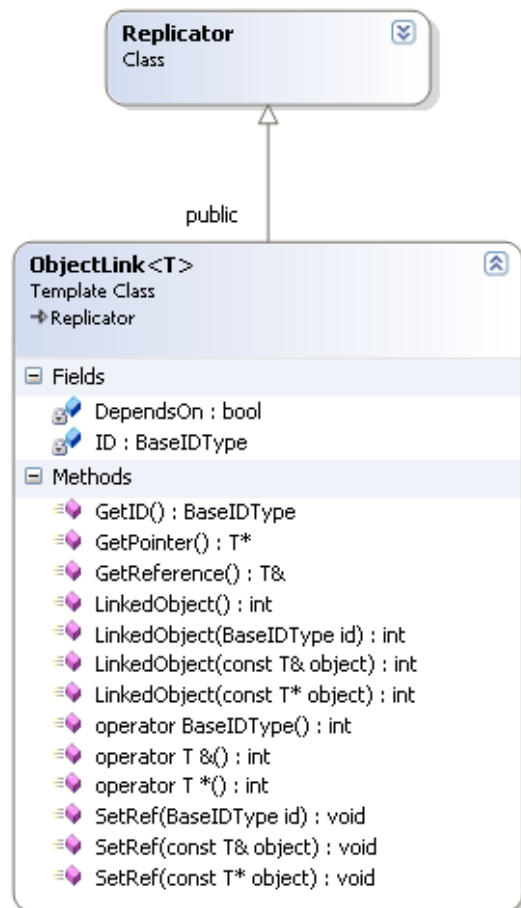


DependsOn() has to be called for each object which depends on another one and RemoveDependencyTo() needs to be called to remove the dependency. If an object is destroyed, all corresponding dependencies will automatically be freed.

The network code will handle the rest of the necessary functionality transparently; that said whenever object B is synchronized it will automatically synchronize object A, too, referring to the example given above.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

The second part comes in shape of a new class:



The ObjectLink template class offers an alternative way of storing references to other objects. Normally one stores either a pointer, a reference or an ID to another object. This pointer/reference/ID can now be replaced by an ObjectLink object. The goal is to automate maintenance of the dependency graph and object (de-)serialization.

To achieve this the class will use the ID of the associated object when serializing/deserializing itself.

If the member variable **DependsOn** is set to true, the linked object will automatically be synchronized along with the object which keeps the reference using the NetworkObject's **DependsOn()**-/RemoveDependency()-methods.

The transfer to the ObjectLink system should be as easy and as transparent as possible. Therefore the ObjectLink class will provide overloaded operators which should allow the usage of that object as if it was a reference/pointer/ID. The only thing which should be necessary to change is the initial declaration and creation of the object and possible code parts which become ambiguous when switching to the new class.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

Using the ObjectLink class can even end in better readable code. Look at the following code snippet:

```
void foo()
{
    BaseIDType myID = A.GetID();
}

void bar()
{
    foo2(myIDMap.GetObject(myID));
}
```

could be replaced by:

```
void foo()
{
    ObjectLink myID = A;
}

void bar()
{
    foo2(myID);
}
```

To complete the system for object references, the game graph's connection system will be adapted to work with the ObjectLink class.

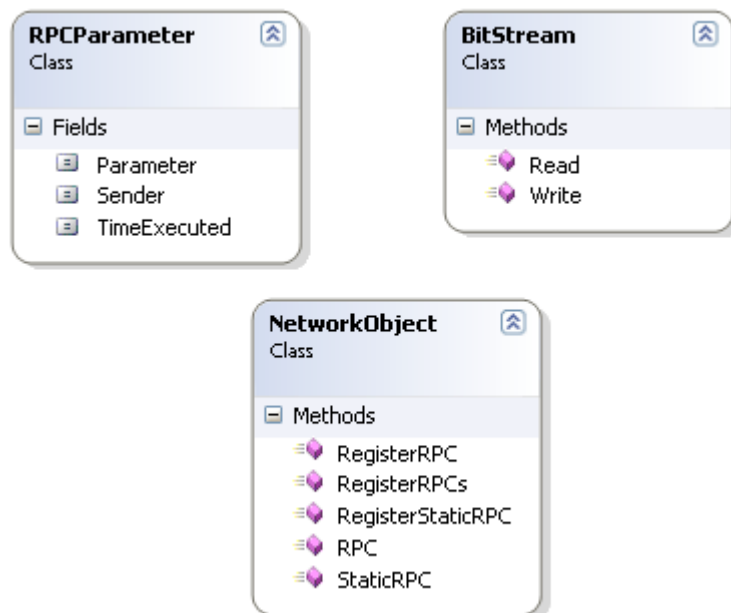
2.5 Remote Procedure Calls

The above mentioned systems are fine as long as we assume that the method calls are solely done on the peer which keeps the ownership of an object. However things become more complicated when performing any functionality on an object which is in proxy mode and the necessary actions differ from those of the ownership mode ones.

Changing functions which need a different behavior depending on the object's role must be handled in individual cases. The network engine provides two helpers to make the job easier. The first thing is the object's Role member variable, which can be used to identify in which role the object currently is. This can be used to distinguish between the two possible object roles.

The second kind of helpers are so called remote procedure calls (RPCs). They provide an easy and intuitive way to remotely call procedures on another machine.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	



The RPC system in our network engine is build around 3 classes. First of all there are some additional methods provided by the NetworkObject class. The RPC()-method and it's corresponding static version allow us to remotely call procedures which have been registered beforehand. That said, if you want to call method A of the corresponding object on the peer which has the ownership you simply call response = RPC("A", parametersbitstream).

The parametersbitstream is an instance of the BitStream class containing the parameter(s) which will be passed along to the remote procedure. Following this call the method A() of the owning peer's object is being called with a single parameter of type RPCParameter. In addition to the passed BitStream, this parameter also contains the ClientInfo of the peer which initiated the method call and the timestamp when it was initiated. The method may then return a bitstream again which will be passed along to the calling peer.

RPCs can also be non-blocking, in which case a callback method can be passed along which will receive the returned Bitstream (if any), specify a priority and flags which define how the remote procedure call is transmitted (unreliable, reliable, ordered, ...).

Before an RPC can be initiated, the method which is to be called needs to be registered. This is being done in two stages. First we overwrite the RegisterRPCs()-method.

In this method each remote procedure is being registered by calling RegisterRPC(&myMethod, "myMethod"), or the corresponding static counterpart RegisterStaticRPC(MyClass, &myMethod, "myMethod").

Second the class is added to a class lookup table, assigning each class a unique string which will be used to identify the class for RPCs on static methods.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

2.6 Event System

We'll add another design constraint to the event system:

EventListener and EventSources only work between objects being in the owner role.

That way we limit the necessary communication paths and ensure that we don't run into trouble in several situations. The constraint can easily be solved by modifying the EventListener and EventSource classes. Attaching EventListeners to EventSource objects is only possible, if the EventListener's role is set to NET_ROLE_OWNER. If it tries to attach itself to an EventSource object which has its role set to NET_ROLE_PROXY, it is automatically passed through to the corresponding owner object using RPCs.

Other changes are made accordingly.

2.7 AI

The AI needs to be altered to be able to transmit any changes to the proxy objects and not run on the proxies at all or at least in another mode (which possibly interpolates the most likely actions on the client and process any incoming PDUs from the server).

Initially this should be possible by simply disabling the AI for all proxy objects.

2.8 UI

With the changes made to the event system as described above, there should be no difficulties with the adaption of the UI.

2.9 Movement Controller

The movement controller needs to be extended, so we can support client side prediction. The necessary changes have been discussed in the section about Dead Reckoning and Client Prediction in the Background document.

2.10 Mission Director

For the mission director the same conditions apply as for the AI. The owning object should have the authority over the state of current missions. The proxy side code should be able to receive updates from the server side mission director.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

2.11 Network Architecture

For establishing a connection, we'll rely on a hybrid client-server architecture meaning. This way we won't have separate server- and client processes but a single process which incorporates both. After the connection has been established the server load (i.e. the ownership of objects) can be distributed along all connected peers.

Because of the planned distribution of simulating the universe, there's hardly any need for providing a dedicated server and therefore we won't separate the server from the client code.

2.12 Object Handover

In order to reduce the workload for a single machine, we add the capability to the system which allows us to transfer the ownership of objects from the server to a client. This feature is vital for being able to sustain a universe in the multiplayer which is at least as complex as the one simulated in singleplayer mode.

One extreme scenario are 6 connected playerships, each flying around in a different city. It's currently expected that simulating even a single city will require almost the complete capacity of a computer, not to mention simulating 6 cities at the same time.

Even if this scenario would be mastered by a high-end PC, the necessary bandwidth to upload the data for all 6 cities could be immense.

This is where object handover comes into play. Instead of having a single server to simulate all 6 cities (which is normally only of importance to the player who is currently inside the city), the client will request gaining control of the city and all related objects. Once he got control over the objects, he'll do the simulation on its own.

2.13 Separate network thread

We'll add a separate network thread. This thread will then run totally independent from the game's frame rate. It can perform operations on the fly (for instance broadcasting chat messages) without having to wait until the main thread is ready. It will also allow us to fine tune the distributed processor time between the two threads to possibly increase the priority of the network thread with the cost of 1-2 fps on the server, to ensure that even a slower server machine does not increase the latency for all connected clients.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

Appendix A - Glossary

bitstream	encoded data in form of a sequence of bits used to reduce the amount of data which needs to be transmitted over the network for instance the data “1” “0” could be encoded into a bitstream like: “100011000”
diff method	The diff method describes a way to reduce the necessary amount of data being transmitted to sent any updates done to an object. Instead of serializing an object and send the serialized stream directly over the net, only the initial serialized stream will be sent. Any following updates will be done by calculating the diff between the last serialized stream and the current one and only send the diff. The receiving client would then reconstruct the serialization stream.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

Appendix B - References

[1] Quazal Net-Z introduction – The Core Innovation: <http://www.quazal.com/modules.php?op=modload&name=Sections&file=index&req=viewarticle&artid=3>

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

Appendix C - Different Approaches

There have been two distinct approaches we came up when first thinking about how a network implementation could look like: the “Synchronized Game Graph Approach” and the “Object Replication and Synchronization Approach”.

The chapters below give account to the basic ideas which stand behind each approach. They are not to be understood as final integration procedures and it should be obvious that neither of the specified ones is able to provide the functionality and features required by a network engine for X4 without significant adjustments to the basic ideas.

Object Replication and Synchronization Approach

This approach has been given different names but they all are based on the same basic idea, which is explained by Quazal for their Net-Z called network framework as follows:

The Object Replication and Synchronization approach “[...] is a type of decentralized, distributed object customized to meet game developers' networking needs. [...] Rather than passing messages from station to station, game objects (such as players, weapons, equipment, monsters, *etc.*) are duplicated to all stations on the network. One station houses the 'duplication master' (the controlling instance) of an object, while all other stations hold a 'duplica' (an exact copy) of the same object. When a master object is updated, the change is then 'pushed' out to all the duplica stations.” [1]

In case of X4 an implementation which strictly follows this approach could look like this:

All objects which need to be synchronized between different hosts in the game define their role on the current machine (which can either be: Authoritative/Owner or Proxy). Initially objects are only created in authoritative mode. Once an object has been created, all connected hosts will be informed of the construction and create proxy versions of this same object. The destruction procedure is handled accordingly.

When the object on the machine which keeps the authoritative role is updated, it synchronizes the objects of all other hosts.

Object synchronization and replication is done on a per class basis, meaning that each class itself defines the necessary methods required to handle incoming synchronization, creation and destruction calls.

Network Integration	Version: 0.9D
Design Analysis	Date: 04/02/08
X4-NET-ANALYSIS	

Synchronized Game Graph Approach

The basic idea of this approach is as follows:

When a client joins the game we need to do an initial synchronization, first. This can either be achieved by sending the complete savegame or only transmit the data which is important for the new client (for instance, information about the sector the player starts in).

After all clients received their initial data package, the transmission mode switches into the per-frame synchronization state, where any changes to the **structure** of the game graph are sent (i.e. changes to the parent/child relationships). This can be a very small dataset per component, and it only needs to include objects that are of a high enough attention level for that client to be interested in. The size of this data can be made smaller using the diff method or by transmitting only changes to the structure explicitly (e.g. add component here, remove here, etc.). The integrity of the data is pretty important.

The next phase is the component synchronization phase. In this phase you prioritize which objects are most important and transmit those first in case we run out of time. For each object we generate an export (a small fragment of a savegame) and then transmit only the parts of that export that have changed (could be a standard diff, could be an XML-specific diff algorithm). Any objects which have not changed (and because of movement controllers etc, there are likely to be many of these as movement doesn't count as "change") don't need any data to be transferred at all.

The final phase would be anything not in the game graph. These could be exported in a similar manner, or by some other means, and their prioritization handled independently of the component prioritization.