I cannot directly process or embed diagrams and charts from Word documents. However, I can provide the English translation of the text content from the document "UniG服务器--设计文档.doc".

UniG Game Server Design Document

Creation Time, Location:
2006-11-29 at U1 Tech
Document Author:
Name	Role	
Document Creation	

Modification Time, Location:

Document Modifier:
Name	Role	Reason	

---

1 Introduction

1.1 Purpose
To elaborate on the overall design of the UniG platform's game server, including the overall game framework, containers and objects, seamless services, map system, processing and Proxy, ILM, and communication with other GameServers.
1.2 Background
Due to the need for communication with Proxy and ILM, correct communication protocols need to be established between them.
As it involves significant network communication and threading content, it relies on the implementation of the base library.
1.3 Definitions (Glossary)
Proxy: Proxy server
ILM: Intelligent Load Balancing center
Seamless Service: Handles cross-server and migration-related operations.
1.4 References
[Should reference ILM and proxy.... design documents] None
2 Overview
The game server process is a large framework, including initialization, driving, and coordinating the work of various modules. Currently, there are the following modules:

- **Server Framework:** Contains the main game framework and base object modules. The main framework drives the entire game process and retrieves and processes messages from the Proxy. The base object system mainly provides a

base class for game world objects, manages them, and implements operations such as insertion, deletion, finding child objects, updating, and sending/receiving messages.
- **Map Module:** Currently, the map module can be understood as a container responsible for managing all objects on the map.
- **Seamless Service Module:** This module is used for load balancing, mainly for handling migration and cross-server processing.

3 Design Prerequisites
None
3.1 Assumptions and Dependencies

- Correct implementation of the base library.
- Good coordination of communication with Proxy and ILM.

3.2 Constraints
[Are there performance indicators, e.g., frame calculations should be completed within 0.05 seconds?] None
3.3 Goals and Guiding Principles

- Provide unified and simple external interfaces.
- Ensure consistency between entity and shadow states in the seamless service module.
- Minimize unnecessary communication volume between GameServer and other communicating parties.
- GameServer, as part of the UniG platform provided to game developers, must have simple, easy-to-use interfaces and facilitate convenient development.
- For the correct and seamless service, ensure the consistency of entity and shadow states.
- Avoid server-to-server synchronous communication blocking the frame calculation process as much as possible; use asynchronous processing more often.
- **Development Method:** Adopt object-oriented software development methods.
- **Architecture Strategy:** In the implementation of the seamless service module, when two objects (shadow or entity) interact, try to use entity data for calculations, then forward the calculation results and control commands to the shadow. Sometimes, for calculation convenience, shadow data can be used directly, with the results forwarded to the entity. Ensure the timing sequence of messages received by an object and its shadow.

4 System Overall Architecture Design
[Diagram placeholder]

The game server process mainly consists of the modules shown in the diagram above. Based on different functional points, we divide them into the following modules:

- Seamless Service Module
- Game Object and Framework
- Map System

5 Server Framework Design Description

5.1 Module Description

The server framework is the driving module of the entire server, responsible for initializing various modules, loading configuration files, starting various threads, and serving as the entry point for all messages. It then executes a loop of retrieving messages, processing messages, and frame updates. The game server process has 6 threads as follows:

- **Main Thread:** Responsible for frame synchronization, sending messages to game objects.
- **ILM Monitoring Thread:** Detects and receives various messages from ILM and puts them into the seamless service message input queue.
- **InfoBus MSG Monitoring Thread:** Detects and receives control messages from InfoBus, such as commands for creating shadows, deleting shadows, forwarding real players to shadows, etc.
- **InfoBus Block Monitoring Thread:** Detects and receives data block messages from InfoBus, such as serialized map data packets for migration.
- **Proxy Monitoring Thread:** Detects and receives messages from Proxy, basically client control messages.
- **Forwarding Session Message Queue Thread:** Each Session has a queue storing messages to be sent to the client. A dedicated thread iterates through these Sessions and sends the messages to the client message output queue.

Additionally, there are four main message queues:

- **Client Message Input Queue:** Stores messages read from Proxy originating from the client.
- **Client Message Output Queue:** Stores messages processed by the GS to be sent back to the client.
- **Seamless Service Message Input Queue:** Stores messages from ILM and Infobus.
- **Seamless Service Message Output Queue:** Stores messages processed by the main thread to be sent back to ILM or Infobus.

5.2 Algorithm

None

5.3 Flow Logic
The game server framework drives the coordinated operation of various modules and is message-driven. The specific operating flow is shown in the figure below.
[Diagram placeholder]
The game framework operation is driven by two types of messages: seamless service messages and game logic messages. The former comes from ILM and adjacent GameServers, the latter from Proxy. Let's first analyze the flow of messages from the proxy.
Sequence diagram as follows:
[Diagram placeholder]
[Is the more detailed diagram we drew together??]
Messages from the proxy message queue first go to SessionMgr. SessionMgr removes the packet header, saves the mapping of ProxyId and Client to Session, and then forwards the message to the Session. The Session then pushes the message to the corresponding player's message queue, waiting for the player to process it.

5.4 Class Detailed Design

- **UniG_GameServer:** The game server object, responsible for initializing the game framework, implementing the frame loop, and cleanup operations before server shutdown.
    - ~UniG_GameServer(): Destructor
    - Initialize(): Initialization function
    - MainRun(): Main program loop function, frame synchronization entry point
    - Finalize(): Cleanup function
    - Update(): Updates all game objects
    - ProcessSeamlessMsg(): Processes seamless service messages
- **UniG_GameObject:** The base object for the game world. All objects in the game must inherit from this class and implement specified interfaces.
    - UniG_GameObject(): Constructor
    - ~UniG_GameObject(): Destructor
    - GetType(), SetType(): Get/Set object type
    - GetName(), SetName(): Get/Set object name
    - GetGuid(), SetGuid(): Get/Set object GUID
    - GetState(), SetState(): Get/Set object state
    - GetPosition(), SetPosition(): Get/Set object position
    - FindObject(): Find an object
    - AddChildObject(), DelChildObject(): Add/Delete child object
    - Update(): Update object
    - InsertMessage(): Insert a message for the object
    - ProcessMessage(): Process a message for the object
    - IsValid(): Check if the object is valid

    - Virtual functions: OnAddChildObj(), OnDelChildObj(), OnUpdate(), OnMessage(), Serialize(), DeSerialize(), Initialize(), Finalize()
- **UniG_Role:** Inherits UniG_GameObject, provides basic role functionalities, currently only interaction with the map, including entering SubMap and accessing the current SubMap.
    - UniG_Role(): Constructor
    - ~UniG_Role(): Destructor
    - Virtual functions: Initialize(), OnUpdate(), OnMessage()
    - EnterMap(): Enter a map
    - GetSubMap(): Get the current sub-map
- **UniG_Player:** Inherits UniG_Role, represents the player in the game, extending UniG_Role with more functionalities. Typically, UniG_Player has a client object and needs to send/receive messages with it.
    - UniG_Player(): Constructor
    - ~UniG_Player(): Destructor
    - Virtual functions: Initialize(), OnUpdate(), OnMessage()
    - EnterMap(): Enter a map
    - GetSubMap(): Get the current sub-map
    - GetSession(), SetSession(): Get/Set the player's session
    - SendMsgToClient(): Send a message to the client

  Note: Developers need to inherit UniG_Player to implement their own player class, handle messages, implement the OnUpdate function. Communication with the client is done via UniG_Session.
- **UniG_Session:** Represents a session, enabling communication between UniG_Player and the client's Player object. Messages from the client are passed to UniG_Player via UniG_Session and stored in its queue. Messages from the server to the client object are sent via the Session. UniG_Session has a message queue for storing server-to-client messages. It also stores the ProxyId and ClientId of the corresponding client for packing and unpacking messages.
    - Fields: m_pPlayer, m_oMsgQueue, m_nServerID, m_nProxyID, m_nClientID
    - UniG_Session(): Constructor
    - ~Session(): Destructor
    - Virtual functions: OnCreate(), OnSocketClose(), OnTimer(), OnReconnect()
    - ProcessMsg(): Process messages from the proxy
    - EnqueMsg(): Enqueue a message into the UniG_Player message queue
    - Front(): Get a message Session needs to send back to the client object
    - Pop(): Delete a message Session needs to send back to the client object
    - FrontBatch(): Get a batch of messages from the queue
    - PopBatch(): Delete a batch of messages

    - SetPlayer(), GetPlayer(): Set/Get the associated player
    - CreateSession(): Static method to create a session
- **UniG_SessionMgr:** Manages UniG_Session instances. It's the unified entry point for processing messages from clients. It unpacks messages, determines the destination Session, and forwards the message.
    - Fields: m_pInstance, m_mapSession
    - Instance(): Get the singleton instance
    - GetSession(): Get a session based on IDs
    - DestroySession(): Destroy a session
    - GetPlayer(): Get a player by GUID
    - ProcessClientMsg(): Process messages from the proxy
- **ProxyListener:** Inherits Thread and ServerSocketHandler, used to accept connections from Sockets.
- **ProxyObject:** Used for communication with Proxy, inherits SocketHandler, implements the OnRead method.

5.5 Interfaces and Protocols

- **5.5.1 External Interfaces:** Users can inherit UniG_GameObject, UniG_Role, and UniG_Player to implement their own game objects and roles. Refer to the detailed design of these three classes for the specific interfaces that need to be implemented.
- **5.5.2 Protocol Description:** The game server framework communicates with Proxy. The data message format from the proxy is as follows:
    - length (unsigned short): Total packet length
    - server id (unsigned short): Target server ID
    - proxy id (unsigned short): Proxy server ID
    - client id (unsigned int): Client ID
    - data (char[]): Message content

4.6 Data Storage
Requires player database access information, TBD.
// TODO
4.7 Constraints
None
4.8 Unresolved Issues
The design of UniG_Player, UniG_Role, and even UniG_GameObject is not detailed enough; it doesn't fully consider what interfaces developers actually need for game development. Additionally, player database loading is not yet implemented, and the module depends on the database module.

6 Seamless Service Module Design Description

6.1 Module Description
The seamless service module is built within the game server process, responsible for encapsulating operations related to seamless cross-server functionality, such as receiving commands from the ILM Server and processing them, receiving data from InfoBus, and returning logic module requests or operation results to corresponding destinations like ILM Server or adjacent Servers.
[Diagram placeholder]
The diagram above shows the communication management between GameServer and surrounding Servers. Since communication occurs between different processes, potentially on different Servers, it's done via Sockets and protocols. The diagram shows three main types of objects communicating with GameServer: ILM Server, Proxy, and other GameServers. The communication part of the seamless service module handles communication with ILM Server and other adjacent GameServers. Additionally, seamless service handles messages related to cross-server transitions and shadow synchronization.
6.2 Algorithm
None
6.3 Flow Logic
The seamless service part is a core component of the platform, primarily handling two aspects:
One: Migration (Moving House)
To achieve load balancing and keep all Servers supporting a world balanced, a migration strategy is needed so players in a Region can correctly move from one Server to another. Migration is as follows:
[Diagram placeholder]
The diagram roughly illustrates the migration process. Details below:
Assumption: ILM Server decides to move map 5 from Server1 (grey) to Server2 (red). Server3 is blue.

1. ILM Server generates migration info based on load (e.g., move map 5 from Server1 to Server2) and sends this command to Server1 and Server2.
2. Server1's ILM monitoring thread receives the command, parses it, pushes it to Server1's main thread message queue, and blocks itself waiting for the result. Server2's ILM monitoring thread receives the command and blocks itself, preparing to receive the migration data block.
3. Server1's main thread retrieves the message, packs the migration information (including all objects in map 5, and objects in 3, 6, 7, 8, 9; maps 1, 2, 4 don't need packing as they are on the target server). Packing rule: If the map being packed is on the same server as the map being moved, pack the objects. If not, pack the shadows generated on the current server for objects on that map. So, pack all objects in 3, 5, 6, and shadows generated on the grey server for objects in 7, 8, 9. It then forms a message and pushes it to Server1's ILM monitoring thread queue.

4. Server1's ILM monitoring thread wakes up, retrieves the message, sends it to Server2, and blocks itself waiting for Server2's acknowledgment.
5. Server2 changes its wait state, receives the migration data packet, unpacks it Region by Region. If a Region belongs to the migrated set, add objects one by one to the current Server (overwrite if a shadow exists). Add objects from other Regions as shadows. Add these Regions to its visible grid set. Send a success acknowledgment back to Server1.
6. Server1 receives the success message from Server2. Server1 requests an MST (Minimum Spanning Tree?) update. ILM updates the MST, notifies all Servers needing the MST, waits for acknowledgments, then tells the affected Servers (involved in migration) to update their shadow lists.
7. Server1 (the source server) receives the shadow update notification, turns objects on map 5 into shadows. Since map 5 moved, its visible range changes, so it deletes shadows for maps no longer visible (e.g., delete shadows generated on this server for map 7). Unlock affected objects on this Server.
8. Server3 (an affected bystander server) needs to update shadow lists for players on maps 7, 8, 9, then unlock.
9. Servers hosting maps 1, 2, 4 are also affected. For instance, players on map 4 might have had shadows on Server1; if 4 is no longer visible to Server1, players on map 4 need shadow list updates. Similarly for players on maps 1, 2, and even 5.

Migration is conducted as a Transaction. The main process of a Transaction is as follows:
[Diagram placeholder]
[Add description of locking maps]
Two: Shadow Synchronization
When an entity or shadow receives a message, it needs to notify the other (shadow or entity) to perform calculations to ensure state consistency. The activity diagram for shadow-entity synchronization is as follows:
[Diagram placeholder]
When an entity receives a message: if it's non-interactive, forward it to the shadow; both calculate simultaneously. If it's interactive, the entity calculates, then forwards control commands to the shadow. The shadow receives the control commands and broadcasts its action to surrounding players. If a shadow receives an interactive message (e.g., being attacked), it retrieves the entity's attributes from the entity (locking the entity; alternatively, use its own attributes temporarily but must lock). After calculation, release the lock on the entity, update the entity's attributes, and send control commands to the entity to play animations.
[Add description of implementing entity locking]
6.4 Class Detailed Design
Three additional threads are started in the seamless service module. One for communication with ILM Server, one for command message communication with adjacent Servers, and one

for data block communication with adjacent Servers. Each thread runs an Epoll loop, listens for socket events, passes them to the main thread for processing, and waits for the main thread's result.

- **ILMListener:** Inherits Thread and SocketHandler, thread for communication with ILM Server.
    - Field: m_pILMEpoll
    - Constructor, Destructor
    - Methods: OnRead(), OnError(), run(), MigrationResponse(), Init(), Handle(UpdateMSTRequestMsg*), Handle(ServerStatusQueryMsg*), Handle(StartPartitionRequestMsg*) (request), Handle(ServerLeaveRequest*), Handle(ServerLockMapRequest*), Handle(ServerUnlockMapRequest*), TransferMigrators(), AcceptMigrators(), UndoMigration(), MsgForward()
- **InfoBus:** Manages connections and communication with adjacent Servers via ServerContext instances.
    - Fields: m_mapServers, m_pMsgThread, m_pBlockThread, m_pMsgEpoll, m_pBlockEpoll
    - Constructor, Destructor
    - Methods: Init(), DeInit(), Instance(), GetSrvCtx(), FindSrvCtx(), CreateSrvCtx(), RemoveSrvCtx(), SetupConnection(), DestroyConnection(), ManageConnection(), SendBlock(), RecvBlock(), AsyncSend(), AsyncSendComplete(), AsyncRecvClose(), AsyncRecv(), SyncSend(), Push(), Args(), SendBatch()
- **ServerContext:** Represents a connection to an adjacent Server. Communication ultimately happens through this class.
    - Fields: m_oServerInfo, m_pMsgConn, m_pBlockConn, m_lstBlockDataList, m_oSendQueue
    - Constructor, Destructor
    - Methods: SetupConntection(), DestroyConnection(), SendBatch(), SendBlock(), RecvBlock(), AsyncSend(), AsyncSendComplete(), AsyncRecvClose(), AsyncRecv(), SyncSend(), GetMsgConn(), SetMsgConn(), ResetMsgConn(), GetBlockConn(), SetBlockConn(), ResetBlockConn(), SetOption(), InitMsgConnection(), InitBlockConction()

6.5 Interfaces and Protocols

- **6.5.1 Interfaces:** None
- **6.5.2 Protocols:** [Complex interactions should have protocol sequence diagrams]
    - **GameServer connection to ILM Server Process:**
        - GS sends connection request (Type 0x0001) with ServerInfo (ID=0, IP, Port, InfoBusPort=0).

        - ILM returns result (Type 0x0002) with status, completed ServerInfo (assigned ID, InfoBusPort), max msg num, batch num, map num, map info list (MapID, Name, PlayerNum, Width, Height, X, Y).
    - **ILM updating GS MST Process:**
        - ILM sends MST info (Type 0x0003) with MST count, MST info list (MapID, ServerID), server count, server info list, adjacent server count, adjacent server ID list, adjacent map count, adjacent map ID list.
        - GS returns result (Type 0x0004) with status.
    - **ILM querying GS status Process:**
        - ILM sends request (Type 0x6001).
        - GS returns result (Type 0x6002) with player num, CPU load, map count, map info list.
    - **ILM notifying GS to start migration:**
        - ILM sends notification (Type 0x6003) with source server, target server, map count, map ID list.
        - GS responds (Type 0x6004) with target server, moved map, result status.
    - **GS requesting exit:**
        - GS sends request (Type 0x6009).
        - ILM responds (Type 0x6010) with result status.
    - **ILM notifying Server to lock maps:**
        - ILM sends request (Type 0x0605) with map count, map ID list.
        - GS responds (Type 0x6006) with result status.
    - **ILM notifying Server to unlock maps:**
        - ILM sends request (Type 0x0604) with map count, map ID list.
        - GS responds (Type 0x6005) with result status.
    - **ILM sending request to Server to update shadows:**
        - ILM sends request (Type 0x4003) with map count, map ID list.
        - GS responds (Type 0x4004) with result status.

6.6 Data Storage
None
6.7 Constraints
None
6.8 Unresolved Issues
None.

7 Map System Module Design Description

7.1 Module Description
The map system can be understood as a container that records map information and information about objects on the map. There are three concepts in the map system: Map,

SubMap, and Region.

- **Map:** Can be understood as the map of the entire world. It doesn't directly manage objects but does so through its child object, SubMap.
- **SubMap:** Can be understood as a piece of the map; the entire world map can be divided into several pieces, each being a SubMap, potentially mapping to a map file. It manages game world objects through Regions.
- **Region:** Can be understood as the specific manager of game objects. It has game logic and manages objects through a logic-independent ObjectContainer.
- **ObjectContainer:** A logic-independent object container, a two-layer mapping: maps object type to a set, then maps object ID to the object. This two-layer mapping improves search efficiency.
- **Cell:** The smallest unit on the map. No dedicated class is designed for it. It records basic information of that cell on the map, such as height, surface type, etc.

7.2 Algorithm
To improve search efficiency, a small algorithm, the two-layer mapping mentioned above, is used. Map type to a set of objects, then map object ID to the specific game object. Essentially, a map within a map.

C++

~~~~
typedef map< TypeKey, KEY2OBJECT* > TYPE2POOL; // Mapping from object type to object collection
TYPE2POOL m_mapTypePool;

typedef map<Key, Object*> KEY2OBJECT; // Mapping from key to object
KEY2OBJECT m_mapObjectPool[nTypeNum];
``` [cite: 29, 30]

**7.3 Flow Logic** [cite: 30]
The logic is relatively simple. Map manages SubMaps, SubMap manages Regions.
Management of game objects is also done through SubMap and then Region. Region
is the actual tool for managing game objects[cite: 30].

**7.4 Class Detailed Design** [cite: 30]
Mainly involves 4 classes. [cite: 30]
[Diagram placeholder - Note: "This diagram looks problematic, seems like an association
~~~~

relationship."] [cite: 30]

* **UniG_Map:** Represents the World concept, manages the map of the entire world[cite: 30].
    * Fields: `m_pInstance`, `m_mapSubMaps` [cite: 30]
    * Constructor, Destructor [cite: 30]
    * Methods: `AddSubMap()`, `RemoveSubMap()`, `GetSubMap()` [cite: 31]

* **UniG_SubMap:** The map structure layer closest to game logic. Logical operations are implemented through `UniG_SubMap` and its derived classes. Mainly manages Regions[cite: 31].
    * Fields: `m_nXRegionNum`, `m_nZRegionNum`, `m_pLeftTopRegion`, `m_pRightBottomRegion`, `m_bLocked`, `m_vecRegionList`, `m_strSubMapName` [cite: 31, 32]
    * Constructor, Destructor [cite: 32]
    * Methods: `SubMapName()` (get/set), `IsLocked()`, `Lock()`, `Unlock()`, `LoadMap()`, `GetXRegionNum()`, `GetZRegionNum()`, `GetRegion()`, `GetCellInfo()`, `Update()`, `InsertObject()`, `RemoveObject()`, `ObjectMoveTo()`, `BroadCastMsg()` [cite: 33, 34, 35]

* **UniG_Region:** The specific class for managing game objects[cite: 35].
    * Fields: `m_nXCellNum`, `m_nZCellNum`, `m_fXUnitLenth`, `m_fZUnitLenth` [cite: 36, 37]
    * Constructor, Destructor [cite: 37]
    * Methods: `Initialize()`, `GetCellInfo()`, `FillCellInfo()` (pure virtual), `FindObject()`, `ObjectEnter()`, `ObjectLeave()`, `BroadcastMsg()`, `IsLocked()`, `Lock()`, `Unlock()`, `AsyncRecvClose()`, `Pack()`, `UnPack()`, `GetRegionLoad()` [cite: 38, 39, 40]

7.5 Interfaces and Protocols [cite: 40]

* **7.5.1 Interfaces:** Users need to implement most virtual functions and all pure virtual functions of `UniG_SubMap`, and also implement some functions of `UniG_Region`[cite: 40].
* **7.5.2 Protocols:** None [cite: 40]

7.6 Data Storage [cite: 40]
None [cite: 40]

7.7 Constraints [cite: 40]

None [cite: 40]

7.8 Unresolved Issues [cite: 40]
None[cite: 40].