Terazona

Version 1.4.1

Developer Guide

(Publication Date: 2004-07-02)

3350 Scott Blvd.,
Santa Clara, CA 95054
http://www.zona.net/

Contents

Part III • Developing With Terazona

List Of Figures

Introduction

Welcome to the Terazona Developer Guide. Terazona provides a reliable and scalable message-passing network infrastructure and a distributed object-view technology for creating persistent, massive multiplayer online game worlds (MMOGs) and simulation environments. The infrastructure consists of a cluster of inter-communicating server that provide validation of incoming client messages and region-based redistribution of validated messages back to clients. Terazona also provides additional game-related services, such as chat- and guild-based message redistribution. Game developers program Terazona using C/C++ client-side and server-side APIs.

Audience

This guide is intended for people who will use Terazona to create massive multiplayer online games (MMOGs).:

Document Conventions

This guide uses a variety of formats to identify different types of information.

Convention	Function
`courier`	Identifies syntax statements, on-screen computer text, and path, file, drive, directory, database, and table names.
`<courier>`	Identifies variable names.
`bold courier`	Identifies text you must type.
italics	Identifies document and chapter titles, special words or phrases used for the first time, and words of emphasis.
underline	Identifies URLs, domain names, and email addresses.
Initial Caps	Identifies Window, menu, command, button, option, tab, keyboard, and product-specific names.
ALL CAPS	Identifies acronyms and abbreviations.
[ ]	Identifies an optional item in syntax statements.
{ }	Identifies an optional item that can be repeated as necessary within a syntax statement.
>	Identifies a separation between a menu and an option.
|	Identifies a separation between items in a list of unique keywords when you may only specify one keyword.

Special Message Conventions

Identifies information that will help prevent system failure or loss of data.

Identifies information of importance or special interest, including Notes and Tips.

Menu Conventions

This guide uses the **Menu > Option** convention. For example, "**Click Format > Style**" is a shorthand instruction for "Click the Format menu, then select the Style option."

Mouse Conventions

To select something, place the on-screen pointer or cursor on the item and click the left mouse button.

To view an **Options** menu, place the on-screen pointer or cursor on an item and click the right mouse button (or left mouse button if using a left-handed mouse). If a menu is available, it will open. (Clicking the right mouse or left mouse button is referred to in this guide as *option-click*.)

When the term *click the mouse on...*is used without qualification, it means to place the on-screen pointer or cursor on an item and click the left mouse button.

To drag something, click the mouse on it and drag the pointer to a different location before releasing the mouse button.

When selecting items from a list using the mouse, you can sometimes select more than one item by holding down the **Shift** or **Control** key while clicking the mouse.

To select a contiguous block of items, click on one item, hold the **Shift** key down, and click on a second item. All items between the two will be selected when multiple selection is enabled.

To select items from different locations when multiple selection is enabled, hold the **Control** key down. Each selected item will remain selected until you complete the action or click the mouse without holding the **Control** key down.

Additional Help

For additional information or advice, contact:

Contact Information United States

Phone	+ (408) 844 9646
Facsimile	+ (408) 844 9647
Internet	http://www.zona.net/
Email	Strategic Partners: strategicpartners@zona.net Developers: gamedevelopers@zona.net Recruitment: jobs@zona.net Information: contact@zona.net
Postal	Shanda Zona, LLC., 3350 Scott Blvd. #23, Santa Clara, CA 95054-3104

Part

I

Overview

This part of the Terazona Developer Guide introduces you to some of the concepts behind Massively Multiplayer Online Games (MMOGs) and Terazona.

Chapter 1

Terazona Architecture

The primary function of Terazona is to provide a lightweight and very scalable network messaging system to support massive multiplayer online games (MMOGs).

Terazona Game Platform Functions

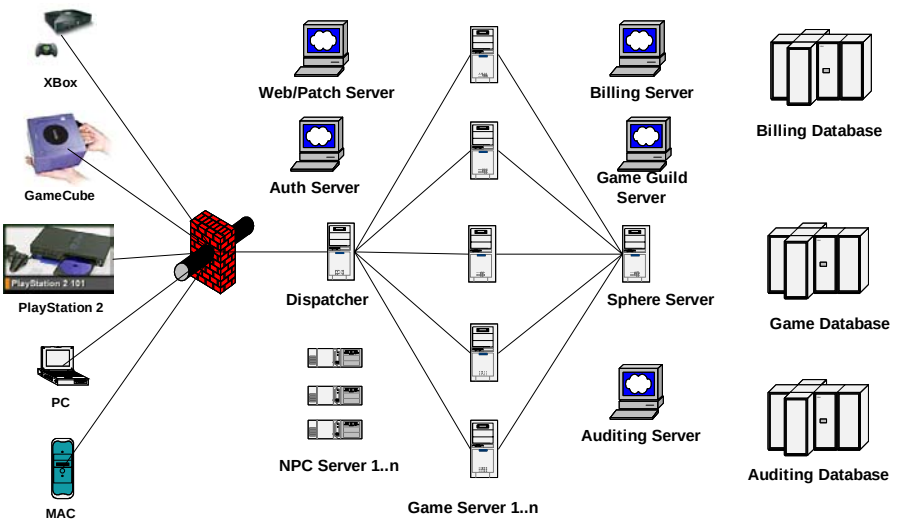
Terazona provides the following game server platform functions:

- A flexible, developer-definable approach to grouping players and objects together. This intelligently determines which game objects receive specific updates. This optimizes the system's use of available bandwidth and processor cycles.
- An API that abstracts away the underlying communication complexities, presenting to developers a simple yet powerful platform model uniquely suited to the development of MMOGs.
- A game-relevant naming system for game development projects that standardizes development effort and promotes cross-functional cooperation within organizations.
- Common and required game functions, such as chat and user profile persistence.
- Reliability through automatic game server fail-over, when deployed in "cluster" mode. Fail-over is invisible to game clients and client dropout does not occur.
- An administrative console that presents a coherent, unified command, control, and monitoring interface for game system administrators.

Terazona Components

Figure 1-1 illustrates the logical multi-tier layout for the Terazona game communications. As developers, you should be concerned mainly with the Game State Servers (GSSs) and the Game Guild Server (GGS) for development purposes. Configuration and cluster management are described in the *Terazona Install Guide*.

Figure 1-1. Logical Network Diagram



Game State Servers

Game State Servers (GSSs) manage Entity State Validaation within game Spheres and game Regions. GSSs manage Entities according to a spatial- or location-based paradigm. Spheres are message group abstractions definable in the server GSAPI API. Sphere grouping is used by the system to intelligently filter out messages to be sent to any player. A game Region is generally defined as the smallest area that a Game State Server will handle within the game. Spheres generally cover multiple Regions within the game. This mechanism makes efficient use of available bandwidth by reducing the number of message packets to be broadcast.

Game State Servers (GSSs) are connection managers for hundreds or thousands of clients. Game-specific logic can be incorporated in the server as a server GSAPI, written in C/C++. Each Game State Server communicates with the ZAC server, periodically sending it game server statistics, such as number of currently logged on players, average message throughput, and so forth.

When deployed in cluster mode, the multiple Game State Servers (GSSs) work in tandem with a Sphere Server. The Sphere Server is responsible for distributing the workload among the Game State Servers.

The Game State Servers also provide other utility services, such as user profile persistence. The server interfaces to supported SQL databases via JDBC, enabling game objects to save their state and persist over time.

Game Guild Servers

Game Guild Servers (GGSs) manage Entity State Updates within Game Guilds. Game Guilds are sets of Entities that share common characteristics and can be used to map in-game design features such as common religions, families, kinship relationships, factions, and clans. GGSs manage Entity State Updates according to a set- or membership-based paradigm. GGSs communicate with each other and with GSSs to keep Entity states synchronized across the Terazona cluster.

The GGS provides various chat functionality, including public chat, private chat, and group chat, and also ensures persistent chat messages. The GGS also provides content filtering and a Chat API (CHATAPI) to perform chat activity validation and management.

Sphere Server

The Sphere Server coordinates and manages Regions across Game State Servers. One Sphere Server will coordinate Region ownership between several Game State Servers. Coordination of spheres across GSSs enables mobile spheres with updates spanning

multiple GSSs. This server enables seamless maps and connected worlds. Sphere Servers are critical to cluster-mode deployment.

ZAC Server

The Zona Administration Controller (ZAC) functions as a centralized process management service in Terazona. All the various processes within Terazona inform the ZAC of their activities. All the Terazona processes establish a link with the ZAC during startup, that is, the ZAC is responsible for validating a process' existence. The ZAC's functionality includes process statistics management, process startup/shutdown validation, and checking for dead processes. Additionally, the ZAC functions as a GUI Client of the ZAC server.

The various processes within Terazona can be activated in any order. All of the server processes -- that is, GSS, Sphere Server, Dispatcher, and Admin tool -- wait for ZAC to be started.

Dispatcher Server

The Dispatcher server is the first point of contact for Terazona game clients. The Dispatcher directs game clients to start a session with a particular Game State Server. It uses an intelligent load-balancing algorithm to determine the least loaded Game State Server which is available to service the requesting client. The Dispatcher periodically receives server load statistics from the ZAC server. It also detects a Game State Server crash and directs the affected clients to connect to another server.

Authentication Server

Players are authenticated in the login stage before character selection. This is performed by the Authentication Server. The Authentication Server provides a security buffer between the client and the database, as well as ensuring that the player is valid.

NPC Server

The Non-Player Character (NPC) server provides a framework for the developer to create and control non-player characters or Artificial Intelligence Entities within the game. This server behaves just like a client, albeit a trusted one. The NPC Controller (a special Entity) can control entire classes of Child Entities such as vehicles, monsters, and so on. NPC Servers provide seamless failover.

Auditing Server

The Auditing Server stores "snapshots" of game activities within an Auditing Database for later retrieval and analysis by Game Masters or System Administrators. We recommend deploying the Game Database and the Auditing Database on seperate database servers to optimize performance.

Message Server

The Message server provides all of the low-end basic messaging between the various servers on the system. They are data agnostic and process all messages between the client and the Game State Servers, as well as being the intermediary between the GSS and the other servers. These are completely transparent in the system, and do not require any knowledge by the developer.

Zona Administrator Application

The Zona Administrator application receives data from the ZAC and displays the server status, as well as allowing the developer to control which servers are running. The user can monitor traffic on the GSS, as well as monitor for error messages.

Zona Modeler

Zona Modeler is not a system component but an XML-based object framework that underpins Terazona. Zona Modeler Objects are the foundation for much of Terazona's functionality and data layer persistence. The Zona Modeler UI enables non-programmatic development and modification of game objects, and during run-time Zona Modeler helps Terazona's servers to exchange bandwidth-optimized game object state change messages and updates.

The C++ Client API

The C++ Client API (CAPI) provides game developers with an interface to write game clients that will work within the Terazona. The API utilizes a custom, lightweight JMS-like messaging protocol. This API is portable across PC, PS2, XBox, and GameCube. Wireless devices will be supported in a forthcoming version.

Game Server Plugin API - GSAPI

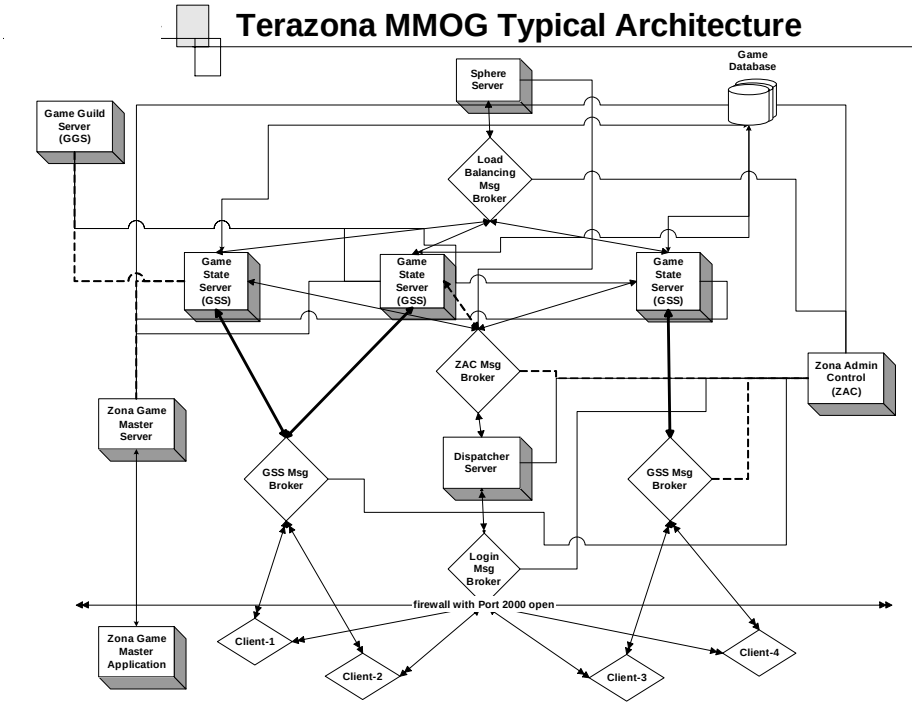
The Game Server Plugin API (GSAPI) is the means to embed game specific logic and validation in the Game Servers. The GSAPI constitutes a system-defined static library,

written in C, in addition to a user-defined DLL. The user-defined DLL, also written in C, uses the functions exposed by the static library to interface with the Game State Servers.

The samples provided with this release illustrate the usage of the GSAPI library. This dynamic design makes it very easy to switch between different game applications, as well as upgrade to newer game logic.

When a GSS receives network messages from the Clients, it passes the messages to the GSAPI library, where they are handled according to developer-specified programming.

Figure 1-2. Terazona Schematic Network Diagram



Chapter

2

MMOG Game Design and Data Flow

The integration of network communications into Massive Multiplayer Online Games (MMOG) is extensive, but it does not necessarily need to be too complex at the game level. This chapter introduces the concepts required to easily integrate the Terazona MMOG engine into the developer's game.

MMOG Concepts

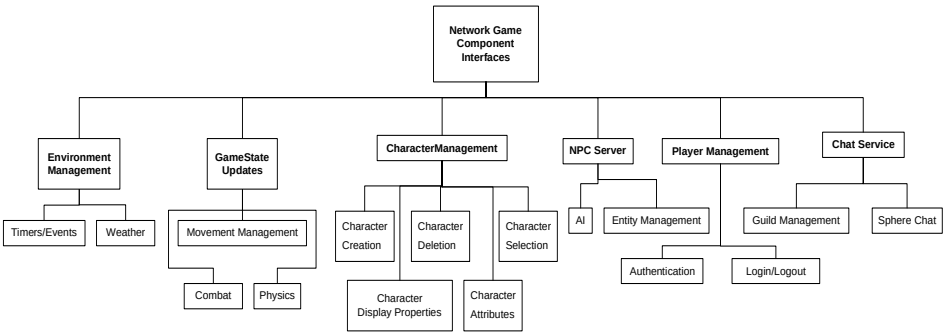
The first thing that needs to be defined are the basic concepts behind the logical components within the game. The game objects are broken down into three separate concepts:

1. **Player** -- Includes all session information about the players;
2. **Entity** --Any network enabled object which exists within the game, including player characters, NPC characters/monsters, and any objects within the game;
3. **Character** -- A special case of the Entity controlled by the Player.

MMOG Network Architecture

This diagram shows the basic logical components for the integration for network communications in a MMORPG.

Figure 2-1. MMOG Logical Components



There are six main elements in this configuration:

1. Game State Validation
2. Character Management
3. Player Management
4. NPC Entity Management
5. Environment Management
6. Chat (Guild Chat and Player-to-Player Chat)

These are by no means the conclusive set of components, nor necessarily a fixed configuration for developing your game. These are mainly guidelines for the simplest integration and a demonstration of how to quickly grasp the concepts behind getting started.

Each of these logical components has elements in both the client side and the server-side GSAPI. We will describe the basic concepts behind each of these logical components as well as a simple design of a character within the game.

Chapter 3

Player Management

Players are the billable components of MMOGs. Players correspond to real-world users who, entering a game, may assume the role of any number of different Characters. This chapter explains how Terazona handles Player login and authentication.

Player Interaction Within Terazona

This is the first step in development of a simple client. This covers the following concepts in the network side of the game:

- Login into the game
- Authentication of the client
- Starting up the Terazona Services on the client side
- Connection to the Game State Server (GSS)
- Entering the player's Character into the game
- Log out of the game and
- Disconnection from the server

Most of this part of the integration starts on the Client side of the game. Several of the components require GSAPI code to enable such functions as user authentication, character selection, and game entrance validation.

Chapter

4

Character Management

Characters are the entities that Players (users) manipulate to interact within a game world. This chapter outlines a typical Character interaction within Terazona.

Character Interaction Within Terazona

Character Management covers all of the control and management of the player's characters within the game. This covers such concepts as:

- Definition of the Character Entity Properties
- Creation of the character
- Selection of the character
- Deletion of the character
- Updating the character
- Validating the character on the server side
- Saving the character to the Database

Chapter

5

Game State Validation

Game State Validation is a central concept within Terazona. Validation preserves the conceptual integrity of game worlds from system errors and malicious hacking. Systematic validation of game messages also optimizes required bandwidth and channels it effectively. This chapter outlines Terazona's approach to Game State Validation.

Understanding Game State Validation

Game State Validation is done entirely on the server-side, within the GSAPI. This is the core functionality of the GSAPI and it handles all of the common game logic. The validation code processes all character and game state updates including such items as position, orientation and damage. It publishes the processed/validated data to those clients within the Sphere of Interest of each client.

Each client receives data from other clients based on its Sphere of Interest. This is defined in the Game State Validation code usually as the Region that the Client currently resides in as well as all of the closest surrounding Regions. This is a single example of one type of Sphere of Interest (SOI). It is by no means the only manner of defining it. The developer is free to define the algorithms for determining each user's Sphere of Interest. These Regions are described in further detail later in this document.

An example of the Game State Validation would be a player moving their character from one position to another. The client sends the character position updates to the GSS through the GSAPI. The GSS validates the data to ensure that it is an allowed move and checks for collision detections. If the move is valid, the data is then published to any other Clients whose Sphere of Interest contains that players' position. If the requested move is invalid, you can direct the GSS to publish corrected data back to the Client.

Game State Validation can also handle such concepts as:

- Player Cheats
- Collision Detection
- Physics
- Combat
- Character Updates (such as health, endurance, and ammunition).

Much of the otherwise tedious data and bounds checking and validation is handled by Terazona's Zona Modeler Framework (ZMO). This automates and optimizes game object creation and modification, handling them transparently and without programmer effort. For more details, see *Zona Modeler on page 29*.

Chapter

6

Entity and NPC Management

NPCs and Entities (objects) within a game world give it flavor and meaning for its participants. This chapter explains how Terazona provides a robust and scalable framework to manage and design interactions between NPCs, Entities, and Characters.

The NPC Server

The Non-Player Character Server handles all the AI for NPCs as well as player to NPC interactions. The NPC Server is treated as a trusted client. It normally resides behind the firewall within the game state cluster. Terazona provides a framework for the NPC Server, and does not limit the developer in any manner. The framework is designed to let the developer easily integrate new NPCs into the game as well as control other aspects of the AI within the game. An XML file is used by the NPC Server to allow for the easy configuration of already defined NPCs for a particular NPC Server instance.

The NPC Server can also control child Entities that a player may own from time to time. Examples of these types of Entities are pets or robotic devices that the player may obtain as they make their way through the game. These Entities are controlled by a Controller - an Entity that manages all Child Entities of a particular Type.

Chapter

7

Environment Management

Game worlds profit from a rich and varied environment that adds realism to the user experience. This chapter explains how Terazona enables easy creation and management of localized and system-wide weather, events, and calamities.

Managing the Environment WIthin Terazona

The environment management is handled by a combination of the NPC Server, the GSSs and the Timer functionality. The NPC Server contains the AI for such things as global and Regional weather and can be generated on the NPC Server by AI code and then it triggers a timer on one of the Game Servers which can then trigger the global timer which updates all of the Game Servers. The Environment management would cover such things as:

- Regional Weather
- Global Weather
- Earthquakes

Chapter

8

Chat Services

One of the requirements of a rich and immersive game world that entices users to return again and again is to facilitate inter-player communication. People want to chat to each other, to create guilds and secret societies, and to recruit like-minded individuals for glorious quests. This chapter describes how Terazona provides a rich, multilayered communicative framework.

Chatting Within Terazona

Terazona provides a complete chat framework for developing chat within the game. This system supports all normal forms of chat such as guild chat, local chat, general game chat and persistent chat. This system can also be adopted for audio inputs such as Microsoft's XBox Communicator.

This is strictly a framework and allows the developer to create their own graphical chat interface. The user may use any or all of the feature sets for their game. For instance some games use persistent chat messages in Guilds, so that if a Player is logged off when a message is sent that message can be saved in the database for later retrieval by the Player.

The Game Guild Servers (GGSs) provide the network infrastructure for Terazona's Chat functionality.

Understanding MMOGs & Chat

MMOGs live or die by the quality and breadth of their chat offerings. A rich chat implementation cannot save a bad game but it is a critical ingredient of every successful MMOG. Instant chat messaging enhances game play, players' tactics, and "feel", while persistent chat lowers account churn by providing a compelling reason for players to return to the game, to renew subscriptions, to engage in Guild politics, and to create elaborate, long-duration avatars to which players develop attachment and into which they invest emotion.

Describing Chat

The Terazona platform provides several key chat solutions:

- Sphere Chat
- Guild Chat

Explaining Sphere Chat

Sphere Chat provides a one-to-one or one-to-many "instant messaging" chat function within Terazona-based games. Players can exchange messages directly with other players' Characters within their Character's Sphere of Interest (SOI). There are several key concepts within Sphere Chat:

Whisper

Normally, a Character's messages are broadcast to all other Characters within the broadcasting Character's SOI. However, a player can make their Character *whisper* to another Character. In this case, the communication is private and not broadcast to other Characters.

Emotes

Players can type or choose a selection of emoticons, or emotes. These are used to convey a feeling (instead of a communication) to Characters receiving the message. For example, instead of a player causing their Character to say "I feel angry", they can activate the sad emote. In this case, other players do not "hear" a message but instead get a textual (or visual) display that says "Character Fnord feels angry".

Auditing

All Sphere Chat messages (their sender, recipient, and content information) can be audited and stored within an Audit Database (AuditDB) for record-keeping and future reference. Sphere Chat messages can be stored pre- or post-filtering. Game Masters and system administrators can examine a specific player's or Character's messages in case of allegations of spamming, stalking, or abuse.

Explaining Game Guild Chat

Guild Chat provides a one-to-one or one-to-many "bulletin board" chat function within Terazona-based games. Players command their Characters to create or subscribe to Guilds and exchange messages with other players' Characters that are members of the same Guilds. Guilds are useful ways for players to organize quests and associations. Guild creation parameters are game-specific; some designers allow all Characters to create

Guilds while others enforce some minimum skill or experience requirement for Guild creation privileges. There are several key concepts within Guild Chat:

Persistent Membership

Guilds can be either persistent or non-persistent membership. When a Character joins a non-persistent membership Guild, their membership lasts only for the duration of that game play session (or until they choose to leave the Guild). If the player logs out and logs back in again, their Character will no longer be a member of that Guild and they must rejoin.

Non-persistent membership Guilds are useful for in-play, transient memberships or for location-specific information. For example, on entering a particular town, a Character can be joined to a non-persistent membership Guild that simulates a town notice board or public newspaper that relays local gossips and town information to them. When they log out or leave the town, their subscription to this information source automatically ceases.

Persistent membership Guilds are useful for long-duration quest-based activities or for in-game player politics. Membership within popular Guilds can become a valuable and prestigious perk for players, while secretive associations such as a Guild of Assassins can provoke fear and excitement within a game community.

Persistent Messages

Guilds can have either persistent or non-persistent messages. With persistent message Guilds, all Guild messages are stored within the Game Database and can be retrieved by Guild members. Messages that arrive while Guild members are offline can be sent to them on-demand after their Character logs back in. In this way, persistent message Guilds function similarly to conventional email.

Non-persistent message Guilds feature transient messages that are not stored within the Game Database. Instead, they are broadcast to all currently logged in Guild members and then flushed from the active system (they are still stored within the AuditDB). Non-persistent Guild members who are offline at the time a message is broadcast will not receive that message.

Moderators

Guilds can have one or more Moderators (Mods). These Mods are a "high-status" Guild member. The Character that creates a Guild is automatically made the initial Guild Mod. Mods can "promote" other Guild members to become Mods. However, a Mod cannot demote other Mods back ordinary Guild member status. Where there is a single Mod in a Guild, that lone Mod can delete the Guild and, similarly, if a single remaining Mod resigns their Guild membership, then that Guild automatically disappears.

Public and Private Guilds

Guilds have “Inviter Attributes” that control the admission rights to particular Guilds. After creating a Guild, players can be allowed to choose how to regulate Guild admissions. Open Guilds will allow any game Character to join without an invite, while closed Guilds enforce Moderator Invitations: a Guild Mod must explicitly send an invitation to a Character offering Guild membership. Mods can therefore restrict membership of Closed Guilds to particular classes or skill levels of Characters.

Extended Guild Features

Game Guilds are not restricted simply to the exchange of chat message content but enable members to share and distribute objects among members and also to alter members’ Character attributes across the entire game world. Game Guild members can share “manna” to cast extra-strong spells, or simple membership of a Game Guild can automatically confer extra wealth or strength or otherwise modify members’ attributes.

Game Guilds enable you to implement Guilds of Sorcerors, Clerics, or Assassins that can increase their members’ powers. This can be very advantageous within an MMOG, and encourages large-scale affiliations and inter-Guild rivalry that improve game playability and “thrill”.

Game Guilds enable Players to communicate with each other using many different channels. They can share strength, manna, karma, luck, or gold, or any in-game object or property. The exact nature and quantity of sharing is defined by game designers, coded by game developers, and validated by the Game State Servers.

Using Game Guilds

Game Guilds are a new technology within Terazona. Although they can provide communication facilities between Players, the older Sphere Chat and Guild Chat services are better suited for transferring text messages. Game Guilds enable players or groups of players to share game objects and modify player abiltiies and inventories over great game distances.

For example, Game Guilds enable game designers to specify an in-game "religion" or "faction" that is aligned to a specific deity. In return for pledging allegiance to this deity, member Characters can receive periodic or permanent alterations to their Character Properties, such as an increase in "manna". "luck", "karma", or any game-specific Character attributes. They can also be used to cast a particular "spell" on all members of that Game Guild.

Game Guilds can feature association references to other Game Guilds, providing game designers with convenient methods to constrain and encourage inter-Guild cooperation and compeittion.

Implementing Game Guilds

Game Guilds utilize a layer of servers within the Terazona cluster known as Game Guild Servers (GGSs) that are additional and orthogonal (though fully connected) to the Game State Server (GSS) layer. GGSs communicate with each other and periodically with GSSs to propagate Game Guild-specific Entity State Updates between Game Guild members.

Filtering Chat Content

Sphere Chat and Guild Chat messages can be filtered at two points: on the server (by game administrators) and on the client (by users or administrators). Using a simple XML "bad words" file, rude, profane, and abusive language can be specified for silent truncation. Players can still create "bad" messages, but other players with content filtering activated will not receive the "bad" content.

Chapter

9

Zona Modeler

Zona Modeler provides an XML-based rapid application development (RAD) tool for creating bandwidth-optimized game objects within networked game architectures. During design-time, Zona Modeler auto-generates Java and C++ code for easy Client- and Server-side integration and deployment. During run-time, Zona Modeler provides a persistence layer by managing complex mappings between in-game objects and database records using object-relational technology. This enables fault-tolerant game world persistence and failover, data security and privacy, and game auditing.

Why Zona Modeler?

Zona Modeler provides a rapid and comprehensive solution to the problem of creating and optimizing networked game objects to communicate and update Entity states across distributed game environments and then persisting Entity states to a database.

Creating optimized network objects without Zona Modeler requires the programmatic development of data structures (such as C++ structs) within the main body of game code. There are a number of problems with this approach.

Game logic becomes intermingled with network game object declaration. Furthermore, there is no central domain where network game objects can be defined and this creates the potential for redundancy and mismatch between NPC Server and GSS game code.

Understanding Zona Modeler

Zona Modeler enables Terazona developers and designers to separate network game object development from game logic development. This decoupling also makes possible an enhanced workflow during design-time where game developers can work on game logic while game designers can work on game object design.

During run-time, Zona Modeler optimizes the bandwidth of the network messages required to propagate Entity state updates. Without Zona Modeler, to update a single property attribute of a game object, a Client would have to send a complete binary object ("blob") across the wire to the GSS. Each subscribed GSS would then have to extract the object from the blob, identify the changed parameter, validate the change, and update its server-side game objects.

Configuring Clients and GSSs to send only the changed values for specific attributes ("entity deltas") required the explicit setting and clearing of dirty, or changed, property attributes. Although this conserved bandwidth and reduced unnecessary messages, it was cumbersome for the developer.

Zona Modeler now handles all entity deltas transparently. Network game objects created or modified within the Zona Modeler graphical user interface (ZMUI) generate bandwidth-optimized C++ and Java code that can be transparently compiled and deployed across the Terazona server cluster and on Clients.

Game developers can focus on game logic and not network logic, while game designers can develop game objects in parallel, non-programmatically, and continually refine and modify them without interrupting game logic code development.

Summarizing Zona Modeler

Zona Modeler provides some very powerful features that enable distributed, heterogeneous transaction state processing:

Zona Modeler enables designers to select certain attributes of game objects as Audit candidates or to select an entire object for Auditing.

Zona Modeler objects not only enable auto-generated, optimized “object deltas”, but also provide transparent and uniform interoperable object layout within the heterogeneous distributed Terazona architecture. For example, game objects generated by Zona Modeler are interoperable between Big Endian and Little Endian systems safely and automatically.

Zona Modeler objects solve type interoperability issues automatically between systems that using various languages running within the Terazona heterogeneous architecture.

Zona Modeler objects also solve someclassic heterogeneous distributed system problems transparently. For example, Zona Modeler enables the easy and automatic identification of remote objects within distributed systems, handling their scoping and lifetime.

Zona Modeler objects support inter-object persistable association and attributes for that association.

Zona Modeler provides features such as the creation of “Interface/Abstraction” objects to enhance model reuse in various model components.

Part II

API Analyses

This part of the Terazona Developer Guide analyzes the design and operation of Terazona itself, and explains how the Client and Server APIs access this functionality.

Chapter

10

Client API Introduction

The C Client API (CAPI) enables game developers to communicate with the Terazona, a complex server cluster environment, without a performance overhead or a requirement to learn distributed computing. The CAPI provides a game-centered suite of powerful functions.

Analyzing the Client API

Client-side entity management is handled transparently thanks to a background, invisible entity caching service called the ZonaEntityManager (ZEM). As a result, you do not need to explicitly manage Entities or their memory allocation or de-allocation. This is handled invisibly by an interaction between the managing GSS and the Client ZEM. The GSAPI provides simple Entity and Character (a special subclass of Entity with a Name) lookup and utility functions. Each GSS maintains a data-driven, event-based record of which Entities it controls, and hence which Entity Services to expose to its managed Clients.

The CAPI provides you with convenience functions that enable you to retrieve Entities by EntityID. The core class is **`ZonaClientEntity`**, and **`ZonaClientCharacter`** inherits from this superclass.

The CAPI consists of a class hierarchy that enables a developer to access all aspects of character game state. There is one main utility function class:

- ZonaServices (subclassed from its parent, BaseServices).

There are five main callback classes that are used to receive Entity data updates, Property updates, event triggers from the managing GSS, and to manage Game Guild and Chat services. These are:

- ChatCallback
- EntityCallback
- GameGuildCallback
- GameStateCallback
- GuildCallback

The ChatCallback and GuildCallback classes are made available for implementing Terazona's Chat interface. They are not required for core Terazona functionality and developers can implement or integrate a custom chat system instead of Terazona's reference chat implementation.

All of the classes are exposed within a Terazona developer installation as C++ header files. The interface files are collected within this directory and its sub-directories:

```
%ZONA_HOME%\include\
```

Understanding ZonaServices

ZonaServices is the main class that provides access to Terazona services. The most important services provided within this class are:

- Player Management
- Character Management
- Property Updates
- Game State Subscription and Publishing

These sets of functions provide many of the capabilities needed for basic game development. Timing Services, Event Triggers, Game Guild, and Chat Management are described in later sections.

BaseServices is the super class of ZonaServices. BaseServices has minimal functionality such as initialization, login and sending and receiving game states. If there was a requirement for a super thin client to run on PDAs, a "lite" library can be used, which includes only the BaseServices and has a smaller footprint.

Understanding ZonaClientCharacter

The ZonaClientCharacter class is very simple. It provides convenient access to the binary data object that comprises the Character Entity Object of a player. ZonaClientCharacter is sub-classed from ZonaClientEntity.

The Terazona system is optimized for fast runtime performance using binary objects. You can use the Auditing Server to extract "regular" entity data and classes and store these in normalized tables. Beneath the CAPI "layer", Zona Modeler autogenerated classes handle the optimization of binary object instantiation and Property update deltas. You use the ZonaClientCharacter class to manage player data logically and simply.

Understanding ChatCallBack

The ChatCallBack class defines functions used for processing incoming Sphere Chat Message data from other clients.

Understanding GameGuildCallback

The GameGuildCallback class defines functions used for processing incoming Guild Chat Message data from other clients.

Understanding EntityCallBack

The EntityCallBack class defines functions used for processing Entity Property updates and enter/exit Sphere activities.

Understanding GameStateCallback

The GameStateCallback class defines functions used for processing the incoming data from other clients.

Chapter

11

GSAPI Introduction

The Game Server Plugin API (GSAPI) enables game developers to communicate with Terazona. The GSAPI's container is a Java Virtual Machine (JVM), but developers write to the GSAPI using standard C/C++ with no requirement to learn or know Java (although a forthcoming version of Terazona will provide a Java API as an option). Additionally, the GSAPI provides a game-centered suite of powerful functions that encompasses Validation, Game State publishing, and data persistence and management.

Understanding Game State Validation

The server GSAPI API is a C++-based API used to create a dynamic link library (DLL) that is loaded by the GSSs and Sphere Servers. When loaded, this DLL is called the GSS Plugin. As the GSSs receive Client updates, individual GSSs pass the messages to their GSS Plugins. Game state updates and Property updates are validated within the GSS Plugins. Likewise, if a collision between moving objects is to be detected on the Server, the GSS Plugin is the place for developers to embed the collision algorithm.

There are several key files:

Table 11-1. GSAPI Key Files

File Name	Description
ZonaPublish.h	Contains functions for publishing game state messages across the Terazona cluster, functions for setting timer events, and various utility functions.
ZonaCharacterValidate.h	Contains functions to validate Client Character management requests (that is, to create, update, or delete).
ZonaEntityValidate.h	Contains functions to validate Client requests for Entity property updates and Child Entity modifications. Also contains functions to validate and monitor entry and exit of Entities and Characters from Spheres of Interest currently managed by GSS.
ZonaGameStateValidate.h	Contains functions to validate incoming Client Game State Data, and to control the updating of replicated Ghost data on other GSSs.
ZonaGuildValidate.h	
ZonaRegionValidate.h	Contains functions to initialize and manage Regions and Entity placement.
ZonaSystem.h	Contains functions to manage GSSs, Clients, and system utilities.
ZonaTimerEvents.h	Contains functions activated by preset event triggers for Entities, Regions, GSSs, or system-wide events.

These header files contain all of the calls used by the Game State GSAPI to handle all of the Game State Validation. These are all virtual functions that the developer can implement to create game logic and respond to Client requests.

All callback functions are prefixed with the "on" modifier.

Implementing the GSAPI

The GSAPI is implemented as a DLL that interfaces with the server through system-defined functions in the **`ZonaServerPlugin.lib`** static library. The name of the resulting DLL should be suitably configured in the **`zona.xml`** configuration file. The Game State Servers (GSSs) as well as the Sphere Servers and the GGS use this file to load their appropriate libraries at startup. When compiled, the same DLL is installed on all Terazona servers. At runtime, the DLL exposes different functionality if instantiated on a GSS rather than the GGS.

For example, the TileTest demo application provided with the release uses a plugin DLL called **`TileTest_ServerPlugIn.dll`**.

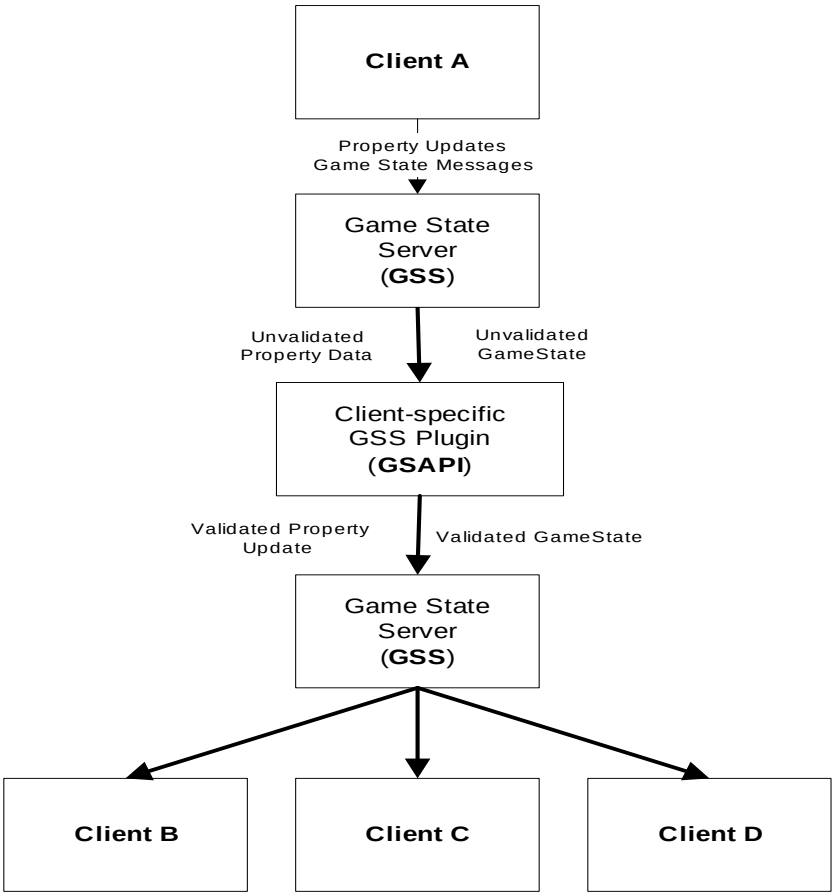
The corresponding entry in the **`zona.xml`** file is:

```
...
<ClusterCommon>
    <LibName authDynamicLibName="AuthSamplePlugin.dll"
                        cryptoDynamicLibName="ZonaCrypto"
       processDynamicLibName="TileTest_ServerPlugIn"/>
...
</ClusterCommon>
```

Understanding Game State Message Flow

The following diagram illustrates the flow of network messages in Terazona:

Figure 11-1. Terazona Network Message Flow



The GSAPI determines valid property updates and messages and forwards them to the messaging subsystem. It can also determine real-time Entity collisions and forwards the relevant collision notification to subscribed Clients. Invalid messages are dropped or negatively acknowledged, and corrected on demand.

The messaging system can then broadcast the message to the appropriate Clients.

The GSAPI uses Regions to determine collisions and perform validations on property and game state updates.

Chapter

12

Regions And Maps API

This chapter explains how to locate and move Characters and Entities within a Terazona MMOG game world using the concept of Regions. It also explains how the Terazona game world can be subdivided into Maps.

Analyzing Regions

The Terazona game world is represented using a pseudo-spatial arrangement of Regions, which roughly correspond to geographical coordinates. All the Regions that immediately surround a particular Region are its Neighbors. Regions use the a proximity paradigm of Neighbors to restrict the broadcasting of Entity state updates only to those Entities that are "neighbors".

An Entity could be in any Region at any given point in time. The game states from that Entity will be shared among its surrounding Regions. As a result, only those Entities presently in neighboring Regions will receive the updates. This is equivalent to standing close to someone and seeing what they are doing. If you stand further away, you will eventually be unable to see what they are doing.

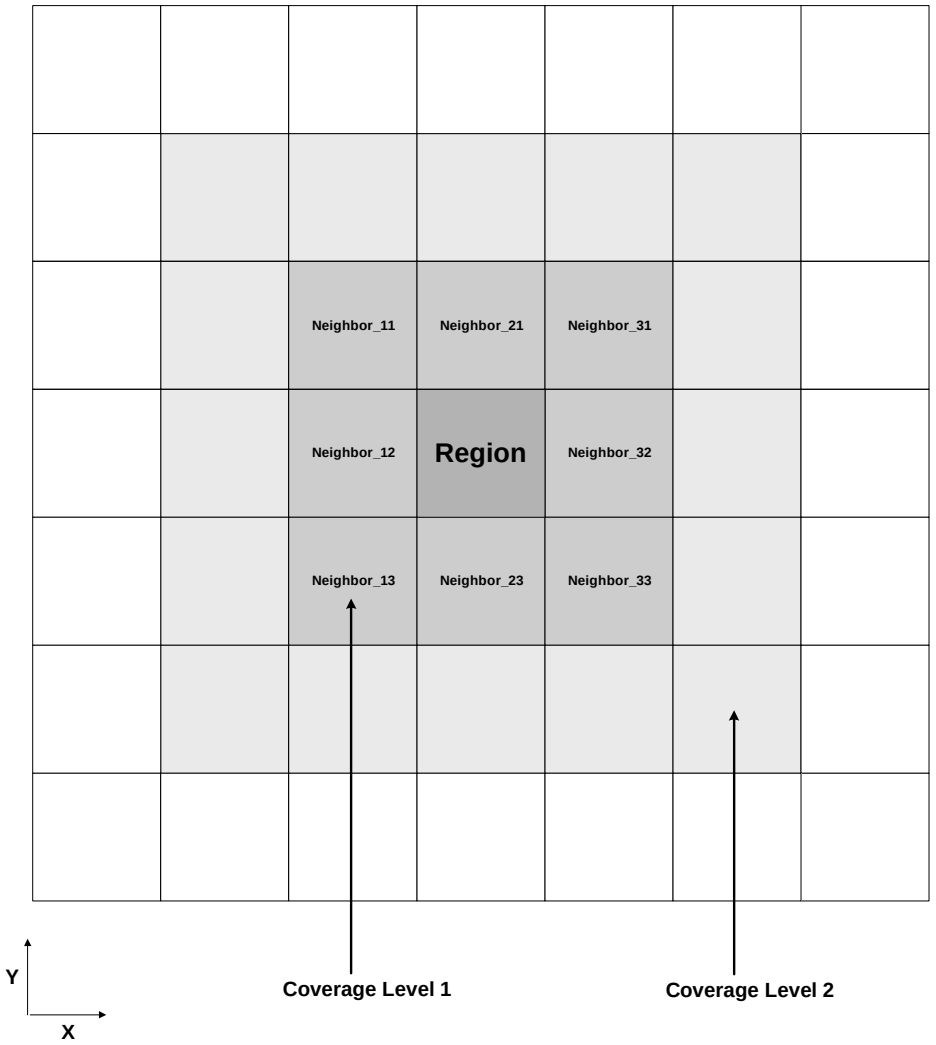
The definition of a Region is completely customizable. The developer is free to implement any algorithm to implement the game world's physical geometry.

For example, a game world could be divided into equally sized spatial grids. Each grid could represent a Region. In this case a Region's neighbors are the eight grids that surround it. This is a classic Euclidean geometry.

Locating Regions

This diagram illustrates this Region concept using the classic Euclidean geometry:

Figure 12-1. Regions and Neighbors



For convenience, the Neighbors are annotated according to their x,y grid coordinate displacement from the central Region. The Region's coordinates in this system would be (2,2).

Analyzing Maps

Maps enable Terazona to load subsets of the entire game world on-demand as Entities travel within specific sets of locations. Maps thus provide a way to do both physical and logical game segmentation.

Maps provide a way to host several different game instances on the same GSS. Players in one Map will be unable to interact with players within another loaded Map. However, both sets of Players could play within the same game terrain and configuration. This enables game designers to segment users into different player cohorts.

Maps also provide a way to subdidive the entire game world into different, independent segments. This provides a way for Terazona to simulate the "shards" or "zones" design found within many MMOG game designs. You can also use independent, dissimilar Maps to provide dungeon "levels" or different floors of rooms or structures. Players can move "between" floors by transferring from one Map to another, and this can be represented in-game using a stairwell/teleporter/elevator metaphor as the Entity is re-assigned to a different Map.

Load-Balancing with Maps

Each GSS loads a single Map or multiple Maps selectively, either during GSS initialization or later, on-demand. Loading each Map causes corresponding data structures for Regions and Neighbors to be created inside the Server plugin.

The notion that each GSS selectively loads each Map into memory on-demand is simple, but its implementation can be as complex or intricate as your game design demands. Game designers and game developers can work together to optimally subdivide the Game World and thus most effectively manage the Map properties and transitions within each GSS plugin.

Designers can, for example, designate a certain set of GSSs to serve a specific portion of the Game World, perhaps one that often experiences high Player demand or where avoiding gameplay slowdowns is most critical. This game-specific tweaking can result in higher scalability, increased Player satisfacvtion, and better load sharing within the Terazona cluster.

Managing Movement

Regions are one of the most powerful features of the Terazona architecture. The Region concept enables game developers to create a game world that completely removes the zoning boundaries from MMOG games. This system also provides the capability to transparently scale the number of players into the tens of thousands for a single virtual world.

As a Character moves around within the game, its current Region location changes. The Region location is validated and constrained by developer-written game logic within the Server plugin. Each Region is 'owned' by a particular GSS. This dynamic ownership is assigned by the Sphere Server and is determined by several factors including the load on each GSS as well as which players are connected directly to that GSS. When two or more GSSs attempt to take ownership of a single Region simultaneously, the Sphere Server arbitrates. This reduces GSS-GSS contention. For more details, see *Managing Region Ownership on page 49*.

Understanding the Sphere of Interest

The player's master Entity Object is stored on the GSS currently assigned ownership of the Region where the Client is currently located. We call these the managing GSS and the owning Client. The Entity Object is also replicated as a "ghost" on all of the GSSs that control any Regions neighboring the Character's Region.

The set of neighboring Regions "around" a Character defines that Character's 'Sphere of Interest', or SOI. This is area in which the Character 'sees' (receives updates from) other players and objects, and other players and objects within the SOI see the Character (if subsribed to game data updates). As other Characters or objects enter this Sphere of Interest, the GSS "owning" those Regions becomes aware of the Character's proximity within its Regions and automatically publishes all other Characters' relevant data to the newly arrived Character.

This process looks like this:

This shows the following layout on each GSS:

GSS1 contains the Player 1 Master Entity Object

GSS2 contains the Player 2 Master Entity Object

As Player 1 moves into an adjacent region to Player 2

GSS1 receives a copy of the Player 2 Ghost Entity Object

GSS2 receives a copy of the Player 1 Ghost Entity Object

As they move apart the replication will be discontinued automatically.

Managing Region Ownership

GSSs feature "sticky" Region ownership. Once it takes ownership of a Region, a GSS relinquishes this ownership of a Region only when all Entities have left the Region and are no longer subscribed to Entity updates from that Region.

The Sphere Server arbitrates all ownership requests between the GSSs, caches the ownership of all Regions, and periodically pushes this data to the other GSSs and the Region monitor for rapid lookup and display.

The Master GSS is the GSS that owns a specific Region and possesses the Master Entity Object for all the Characters and Entities (including all Child Entities) in that Region.

The Ghost GSS is any GSS that possesses a Ghost Entity Object of any Character, Entity, or Child Entity.

A Local Entity is an Entity located within a Region owned by the same GSS that currently possesses the Master Entity Object for that Entity. Local Entities' Property and Game State updates are validated and modified directly by their local, Managing GSS.

A Remote Entity is an Entity located within a Region owned by a GSS that currently possesses a Ghost Entity Object for that Entity. Remote Entities' Properties and Game States are modified by their local GSS upon the reception of update requests from the Master GSS.

The GSS that owns a specific Region processes all Property updates and game state validations within that Region, and propagates the Entity state changes to all other GSSs subscribed to updates from that Region.

During login, the Dispatcher assigns Clients to the least-loaded GSS. When that GSS relinquishes ownership of the Client (that is, it changes from that Client's Master GSS to a Ghost GSS), then although that GSS continues to communicate with that Client, it forwards for validation all traffic for that Client to the GSS currently that Client's Master GSS. In effect, the original GSS functions only as a message router.

Terazona's dynamic Region ownership is a key element that allows Players to experience a seamless world. By caching the Entity state information using master and ghost copies, rapid promotion and demotion of Master and Ghost Entity Objects, and by ensuring that the active Entity state validation relocates on-demand to least-loaded GSSs, Terazona provides a transparent way for game developers to create enormous worlds without user-perceptible boundaries or loading lag.

Summarizing Regions

The Master Entity Object is located on the GSS that owns the Region where the Client is currently located.

The Ghost Entity Object is replicated on every GSS that owns any of the neighboring Regions where an Entity has subscribed to Property updates and Game State Messages.

The GSS is responsible for all collision detection and messages broadcast for all Entities within the set of Regions that it owns.

Each GSS requests ownership of a Region when one of its Managed Clients enters an empty, unowned Region. A Managed Client's Master Entity Object is located on its Managing GSS.

Each GSS relinquishes ownership of a Region when all the Entities have left the Region. The GSS tells the Sphere Server that its no longer interested in that Region.

The Sphere Server is the arbitrator when two GSS requests for ownership for the same Region at the exact same instant.

Each GSS caches ownership information for every Region in the universe in a lookup table.

When a Client sends its data to a GSS, if the Region where tha Client is located is owned by that GSS, then that GSS processes the Client data. If not, the GSS forwards the Client data to the GSS that currently owns the Region where the Client is located.

Understanding the Master/Ghost Entity Relationship

The Player's Character data is copied to the GSSs that control the neighboring Regions. This is a Master / Ghost relationship.

The Master controls all of the updates of the character's Entity Object to the surrounding Ghost Entity Objects.

Updating Ghost Entity Objects

The Ghost Entity Objects get updated in the following manner:

1. The Master GSS Plugin calls **`set`*`XXXX`*`()`**.
2. This call causes the Master GSS to send messages to any Ghost GSSs that contain a Ghost Entity Object of the Master Entity Object.
3. These Ghost GSSs then trigger the Ghost Entity Object update callback function **`ZonaEntityValidate::onNotifyEntityUpdate()`**. Any Clients managed by this GSS and that have subscribed to Entity updates will receive an **`EntityCallback::onNotifyEntityPropertyUpdate()`** callback and update their local Entity data. **Entities that are members of Guilds associated with the publishing Entity can also receive Entity State Updates.**

Entering a Sphere and Exiting a Sphere

As a player's Character moves between game Regions their 'Sphere of Interest' (SOI) moves with them. As one player's Character moves within other players' SOIs the GSS calls the **`ZonaEntityValidate::onEnterEntity()`** and **`ZonaEntityValidate::onExitEntity())`** callbacks on the Plugin to prompt the Plugin to automatically push the new data out to subscribed Clients.

Table 12-1. Sequence For Client A Entering Client B's Sphere

Client	GSAPI	GSS
Player A enters Player B's sphere		
		`ZonaEntityValidate::onEnterEntity()` message is sent to the Plugin.
	`ZonaEntityValidate::onEnterEntity(){` developer code here }	
`EntityCallback ::onNotifyEntityJoinedSphere()` called on Client.		

When player A exits Player B's Sphere, player B is informed that player A has left the Sphere. This sequence is outlined below:

Table 12-2. Sequence For Client A Exiting Client B's Sphere

Client	GSAPI	GSS
Player A exits Player B's sphere		
		`ZonaEntityValidate::onExitEntity()` message is sent to the Plugin.
	`ZonaEntityValidate::onExitEntity(){` developer code here }	
`EntityCallback ::onNotifyEntityDepartedSphere()` called on Client.		

Managing Maps

Terazona's APIs enable you to manage disjoint, distinct sets of Regions as a Map. Each Map is distinguished by a **`MapId`** integer.

Initializing Maps

During startup (or when an Entity moves into an unowned, unloaded Region) the GSS calls **`loadMap()`**. This causes an **`onLoadMap()`** callback to fire within the Server Plugin. You should place your initialization code to execute this function:

```
bool onLoadMap(int mapId)
```

You should also notify Terazona within the Server Plugin of the size of the in-memory Map by calling this function:

```
int getRegionCount (int mapId)
```

You then must also notify Terazona within the Server Plugin of the dimensions of the in-memory Map by calling these functions:

```
int getRegionDimensionX(int mapId);
int getRegionDimensionY(int mapId);
int getRegionDimensionZ(int mapId);
```

Managing Maps

Each Entity's MapId must be set by the Server Plugin in the **`ZonaEntityValidate::onEntityJoinedGame()`** callback:

```
int onEntityJoinedGame(ZonaServerEntity* entity);
{
    entity->setMapId(0); // assign  a mapId to the entity
    return 0; // return whatever region the entity
              // has to start in the corresponding map
}
```

After this, the Server Plugin can trigger a map change for an Entity by setting the Entity's **`mapId`** within any callback.

You can call this function as required:

```
void getRegionNeighbors(int mapId, int rgnId)
```

Checking Neighbors

You can check neighbors using the callback **`ZonaRegionValidate::`**

```
int* onGetRegionNeighbors (int mapId, int regionId,
```

```
byte regionCoverage, unsigned int* numIds)
```

Monitoring Maps and Regions

The Region Monitor process attaches to the Sphere Server and can display a grid illustrating the current Map/Region usage. It uses the **`getMapDimension`*`X|Y|Z`*`()`** calls.

Publishing Data Using Functions

As the Characters move around in the game, they must publish their new property and game state data to the GSS. There are several API function calls available for different publishing requirements.

Each of these calls publishes specific data to other GSSs, other Clients, and the game and audit databases.

Table 12-3. Data Publishing Functions

Data Publishing Functions	Description
`ZonaServices:: publishEntityPropertyUpdates()`	Publish Dirty Properties for a specific Entity.
`ZonaServices:: publishAllEntityProperties()`	Publish all Properties for a specific Entity.
`ZonaClientEntity:: publish()`	Publish calling Entity's Dirty Properties.
`ZonaClientCharacter:: publish()`	Publish calling Character's Dirty Properties.
`ZonaGSPublish:: publishGameStateMsgInSphere()`	Publish a Game State Message to all subscribed Entities within the calling Entity's SOI.
`ZonaGSPublish:: publishGameStateMsgToEntity()`	Publish a Game State Message only to a specified Entity. The Client hosting the Entity receives the message, even where that Client did not subscribe to Game State Messages.
`ZonaGSPublish:: publishGameStateMsgTo AllMasterEntities()`	Publish a Game State Message to all Master Entities. This is a convenience function that calls `ZonaGSPublish:: publishGameStateMsgToEntity()` internally.

Note that the Game Database and Audit Database are only updated following a call to `ZonaServerEntity::save(bool audit)`. Calling this function causes the Game Database to be updated, while the boolean option controls whether or not the Audit Database gets updated.

Receiving Data Using Functions

There are corresponding callback functions for updating the Entity Object with new, published data.

Table 12-4. Data Reception Functions

Data Reception Functions	Description
`EntityCallback:: onNotifyEntityPropertyUpdate()`	Called by ZonaServices to notify the Entity of a Property update.

Table 12-4. Data Reception Functions

Data Reception Functions	Description
`EntityCallback:: onNotifyEntityParentChange()`	Called by ZonaServices to notify the Entity of a change in its Parent Entity.
`EntityCallback:: onNotifyEntityJoinedSphere()`	Called by ZonaServices to notify the Entity that another Entity has entered its SOI.
`EntityCallback:: onNotifyEntityDepartedSphere()`	Called by ZonaServices to notify the Entity that an Entity has exited its SOI.
`GameStateCallback:: onReceivedGameStateMsg()`	Called by ZonaServices to notify the Entity of a change in Game State Data.

Chapter

13

Physics and AI API

Physics and artificial intelligence is a key component of a realistic MMOG. This chapter describes how to embed such realism within Terazona.

Detecting Collisions and Simulating Physics

Collision detection is normally validated on the GSS using through the GSAPI. The Client should handle its own collision detection, but the GSS usually validates any collision detection occurring between characters and other characters or NPCs. This is not a requirement, merely a recommendation. Developers are free to implement all levels of collision detection and physics wherever they wish. Tuning a system for performance and load-balancing, however, requires careful game design.

Using Artificial Intelligence

AI is normally handled completely by the NPC Server. The NPC Server functions as a Trusted Client within the Terazona cluster.

Managing Items

Most of the item management is handled by the NPC Server, but we will go into the some of the server-side specifics of item management. The NPC Server is completely developer definable. We do provide an example framework that works well for handling all of the items and Non-Player Characters within the game.

Chapter

14

NPC Controller API

This chapter explains how to control Non Player Characters (NPCs) using the NPC Server.

Managing Items

Most of the item management is handled by the NPC Server, but we will go into the some of the GSAPI side specifics of item management. The NPC Server is completely developer-configurable. The Terazona installation provides an example implementation that works well for handling all of the items and Non-Player Characters (NPCs) within a game.

In order to describe how Terazona handles item management, this chapter describes some of the basic concepts behind the NPC Server.

Understanding the NPC Controller

The NPC Server is a server-side Client that manages Non-Player Characters (NPCs) and game items.

Like Player Clients, the NPC Server uses the Client API (CAPI) library to interact with other Clients in the game world. For security, NPC Servers should be deployed behind the firewall.

A game deployment may have several NPC Servers. Multiple NPC Servers can provide greater performance, as well as redundancy and fail-over capability during single NPC Server crashes.

Creating the NPC Controller

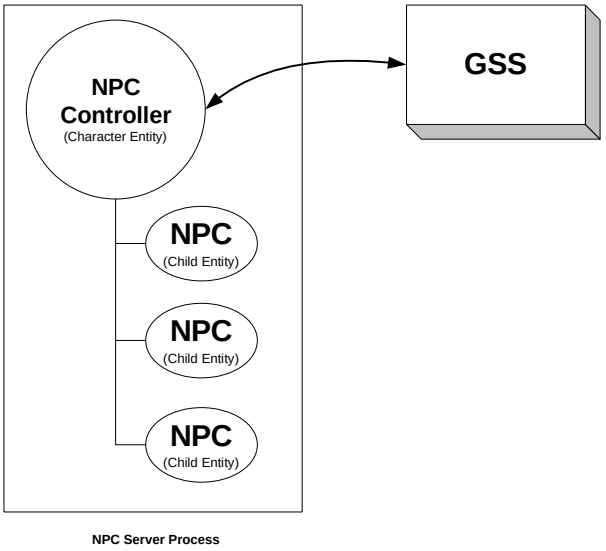
Each NPC Server logs into the Terazona cluster as a normal Client. Although every NPC Server may login to several ZonaServices, only one login per NPC Server is recommended. The Character Entity used for logging in the NPC Server is called the NPC Controller.

Create a user entry in the database for the Controller to describe the user role. During the very first login, the NPC Server creates the Character Entity. During this Character creation, the Entity is set to **`ZONA_ET_IS_HIDDEN`**. This prevents other Clients from receiving any Controller game states.

Managing Child Entities

The NPC Server controls all NPCs, such as monsters and other game items (not controlled by a Player's Character). Each of these items becomes a Child Entity of the NPC Controller:

Figure 14-1. NPC Server Process Schematic



Individual NPCs, if mobile, can roam to any permitted location within the game world. The associated NPC Server thus receives all the game states within the locale of each of its owned NPCs.

Tuning NPC Server Performance

Because each NPC Server will handle several items and NPCs that are spread out across several (possibly disjoint) Regions, NPC Servers can receive large quantities of messages. Therefore, good game design can mean distributing NPC Servers "across" the game world that control specific Regions clusters, or areas.

Managing NPC Servers

To monitor the status of NPC Servers via the Zona Administrator, developers should subclass and implement the CAPI BaseServer class:

```
BaseServer.cpp.
```

This enables monitoring, startup, and shutdown of NPC Servers. Developers implement the BaseServer functions provided to enable NPC Server control.

Chapter 15

Timers API

This chapter explains how you can set Timers to control game world events.

Understanding Timer Events

Terazona includes a Timer Event Functionality that fires events on the Server. Clients (either Player's Characters or NPC Entities) send requests to their GSSs to set triggers that will activate based on local or system-wide clocks. The GSSs then fire the events at the set delay, activating. This can be used in many different ways, such as creating events that happen at specific times in the world, Players' birthdays, anniversaries, weather changes, or recurring events such as a geyser going off or an entertainment show happening.

There are four different types of Timers:

1. Entity Event Timer
2. Region Event Timer
3. GSS Timer
4. World Timer

The timers can be set as recurring timers. The timer 'period' value determines the recurring nature of the timer while the delay is set in milliseconds. For example, a 'period' value of 3, would make the event fire 3 times.

To set infinitely repeating timers, set the period value equal to **`-1`**.

All the timers are set within the GSS Plugin using one of the **`ZonaPublish::set`*`XXXX`*`LevelTimerEvent(...)`** functions. All the timers return a Task ID for the timer event (and a **`-1`** if unsuccessful).

All the timers fire a callback within the GSS Plugin using one of the **`ZonaTimerEvent::on`*`XXXX`*`TimerEvent(...)`** functions.

You can cancel previously set timers using the **`ZonaPublish::cancelTimerEvent(int eventId)`** function.

Using Entity Timers

The **ZonaPublish.h** call is:

```
int setEntityLevelTimerEvent(int entityId, long delay,
                    byte* eventData, short size, int period)
```

An Entity-level timer associates with a particular Entity. Entity-level timer events fire in the GSS that owns the Entity (that is, the GSS that contains the Entity with an **isMaster** status set to **true** when the event triggers). Therefore, even if one GSS sets the timer, the event may be fired in a different GSS that is managing that Entity at that time. If the Entity logs out, all associated events will still fire. In these situation, the **ZonaTimerEvents:onLoggedOutEntityTimerEvent()** callback fires instead of the **ZonaTimerEvents::onEntityTimerEvent()** callback.

Used for time-based spells on specific Characters or Entities. For example, werewolf or shapeshifter spell and periodic powerups.

The associated **ZonaTimerEvents.h** callback is:

```
void onEntityTimerEvent
                     (int eventId, ZonaServerEntity* entity,
                       byte* eventData, short eventDataSize)
```

Using Region Timers

The **ZonaPublish.h** call is:

```
int setRegionLevelTimerEvent (ing mapId, int regionId,
         long delay, byte* eventData, short size, int period)
```

A Region-level timer associates with a particular Region. Region-level timer events fire in the GSS that owns that Region at the time the event triggers. Therefore, even if one GSS sets the timer, the event can be fired in a different GSS. If the Region does not have any owner when the event is due to trigger, then the event is cancelled.

If a tree falls in Terazona's forest and nobody is there to watch it fall then it does not make a sound!

Used for game-specific traps, triggers, localized weather conditions, or spells associated with certain temples or tombs.

The associated **ZonaTimerEvents.h** callback is:

```
void onRegionTimerEvent (int eventId, int mapId,
                   int regionId, byte* eventData, short size)
```

Using GSS Timers

The **ZonaPublish.h** call is:

```
int setGSSLevelTimerEvent
       (long delay, byte* eventData, short size, int period)
```

A GSS-level timer is associated with the local GSS that sets the timer. GSS-level timer events fire only in the local GSS Server plugin that set that timer. If that original GSS shuts down, all GSS-level timer events specific to that GSS are lost.

You can use this timer as a generic timer for internal GSS system maintenance tasks, database cleanup, cheat analysis, memory cleanup, synchronization, and so on.

The associated **ZonaTimerEvents.h** callback is:

```
void onGSSTimerEvent
                  (int eventId, byte* eventData, short size)
```

Using World Timers

The **ZonaPublish.h** call is:

```
int setWorldLevelTimerEvent
       (long delay, byte* eventData, short size, int period)
```

A World-level timer associates with all the GSSs within the Terazona cluster. World-level timer events fire in the local GSS Server Plugin that set the timer as well as all the other GSSs and the Sphere Server. If the GSS that sets the world timer shuts down, the events will still fire in all other, active GSSs. When a GSS comes online, it synchronizes with all pending world events.

Developers can use this timer for system-wide state synchronization. This can be used to affect Entities on a system-wide basis. For example, you can use World-level timer events to create a global in-game clock time.

The associated **ZonaTimerEvents.h** callback is:

```
void onWorldTimerEvent
                  (int eventId, byte* eventData, short size)
```

Chapter

16

Chat Client API

The Chat Client API enables game developers to initiate and respond to Chat events within Clients and to provide Chat services for players' Characters. Chat is managed using a combination of function calls that are validated by a Client's managing GSS and function callbacks that receive validated Chat data from the Terazona servers. Chat is transmitted through the GGS network.

Analyzing Chat Message Structure

Chat Messages are the basic unit of communication within the Terazona Chat Framework. A single Chat Message is encapsulated within the **`ChatMsg`** class. Clients exchange these serializable Chat Messages with other Clients by sending them across the wire to the GGS for validation. Once validated, they are dispatched back across the wire to the designated recipient Clients, which deserialize the incoming Chat Messages and respond using a hierarchy of Chat Handlers.

ChatMsg

The **`ChatMsg`** class is used for direct Character-Character Sphere Chat and its definition is found here:

```
%ZONA_HOME%\include\message\data\ChatMsg.h
```

In addition to a default blank constructor, the parametrized **`ChatMsg`** constructor enables the message creator to specify the Recipient ID, the intensity level of the message (that is, whether recipients perceive you as SHOUTING or merely talking), and the message body.

Within every ChatMsg there are 9 standard attributes:

Table 16-1. ChatMsg Attributes

Attribute	Purpose
`LONG64 date`	Specifies the date the message was sent.
`int flag`	Specifies the status of the message, using enumerated **`MailFlags`**.
`int senderId`	Specifies the Entity Id of the sending Character.
`int recipientId`	Specifies the Entity Id of the message recipient (the Entity Id can denote either a Guild or a Character).
`char* toCharacterName`	Specifies an optional recipient Character Name.
`char* subject`	Contains the message subject, in Unicode.
`char* body`	Contains the message body, in Unicode.
`unsigned char intensity`	Specifies the message intensity. Although currently not used within the Terazona Chat Framework, the value is passed through to Clients so developers can implement custom intensity UI handlers.
`int recipientProperties`	Identifies the **`ChatType`**, that is, the type of Chat Message. Used when registering a handler callback for a specific type of Chat Message.
`int id`	Defines a persistent Id for each Chat Message.

The class definition also contains standard functions to serialize and de-serialize the messages. In addition, there are several special values enumerated within the ChatMsg class. The **ChatMsg** Mail Flags enable developers to create email-style messaging applications for Clients.

Table 16-2. ChatMsg Enumerated Values - MailFlags

Enumeration	Value	Purpose
FLAG_UNDOWNLOADED	1	If set, indicates this message has not yet been downloaded.
FLAG_UNREAD	2	If set, indicates this message has not yet been read.
FLAG_DELETED	4	If set, indicates this message has been deleted.
FLAG_REPLIED	8	If set, indicates this message has been replied to.

The **ChatMsg** Recipient Id values are used to control message distribution.

Table 16-3. ChatMsg Enumerated Values - SpecialRecipients

Enumeration	Value	Purpose
RECIPIENT_ID_SPHERE	-1	Message will be received by all Characters in the sender's SOI.
RECIPIENT_ID_SYSTEMWIDE	-2	Message will be received by all Characters throughout the game world. Because all messages require GSS validation before propagation back to Client, permission to use this "broadcast" feature of Sphere Chat can be reserved for game masters or system administrators.
RECIPIENT_ID_BY_NAME	-3	Message will be received only by specified Character.

Chat Intensity constants are defined within this file:

```
%ZONA_HOME%\include\zaf\ZonaGlobalConstants.h
```

Table 16-4. ZonaGlobalConstants - Chat Intensity

Definition	Value	Purpose
CHAT_INTENSITY_WHISPER	1	Sets Chat Intensity to "whisper"
CHAT_INTENSITY_TALK	2	Sets Chat Intensity to "talk"
CHAT_INTENSITY_YELL	4	Sets Chat Intensity to "yell"
CHAT_INTENSITY_SHOUT	8	Sets Chat Intensity to "shout"

Chat Message Types (used as **ChatType** in the **recipientProperties**) are also enumerated within this global constants file:

```
%ZONA_HOME%\include\zaf\ZonaGlobalConstants.h
```

ChatMsgPtrVector

The **ChatMsgPtrVector** class definition is found here:

```
%ZONA_HOME%\include\message\data\ChatMsg.h
```

This is vector of type **ZVector** that contains a list of **ChatMsg** objects. The **ChatMsgPtrVector** is a container for the incoming and outgoing **ChatMsg** objects that provides standard vector modification and iteration functions.

The **ZVector** class definition is found here:

```
%ZONA_HOME%\include\util\ZVector.h
```

ZonaGuildChatMsg

Unlike Sphere Chat, Guild Chat uses this function to send messages to other Guild members:

```
GameGuildServiceInterface::
                sendChatMessage (ZonaGuildChatMsg* aMsg)
```

Sending Chat

Every instantiated **`ZonaClientCharacter`** gains access to the client functions provided in **`ZonaServices.h`**. The function for sending Chat Message payloads is **`sendChatMessage()`**.The steps involved in sending a Sphere Chat Message are:

1. Instantiate a ZonaServices object:

   ```
   m_zona = new ZonaServices(true, true);
   ```

2. Create a suitable **`ChatMsg`** using the **`ChatMsg::ChatMsg(...)`** function. You must supply three parameters:
   a. The Recipient Id, defined as an **`int`**. Choose one of the following:
      - An Entity Id (or a corresponding Guild Id)
      - The enumerated value **`RECIPIENT_ID_SPHERE`** (-1)
      - The enumerated value **`RECIPIENT_ID_SYSTEMWIDE`** (-2)

   b. The Intensity Level, defined as an **`unsigned char`**. Choose one of the following:
      - **`CHAT_INTENSITY_WHISPER`**
      - **`CHAT_INTENSITY_TALK`**
      - **`CHAT_INTENSITY_YELL`**
      - **`CHAT_INTENSITY_SHOUT`**

   c. The Message Body, defined as a **`char*`**.
   d. The complete function invocation should look something like this:

      ```
      ChatMsg msg(myRecipID,CHAT_INTENSITY_TALK,myMsg);
      ```

3. Send the Chat Message by calling the **`ZonaServices::sendChatMessage(msg)`** function:

   ```
   m_zona->sendChatMessage(msg);
   ```

The Sphere Chat Message has been sent by the Client. Following Server-side validation, it will be broadcast to the destination Clients and received by them using callback Chat handlers.

Sending Guild Chat

Sending Guild Chat messages is fundamentally similar to sending Sphere Chat messages, except that the recipient parameter is a Guild Id (that is, an Entity Id that specifies a Guild). The steps involved in sending a Guild Chat Message are:

1 Instantiate a ZonaServices object:

```
m_zona = new ZonaServices(true, true);
```

2 The Character must already be a member of the Guild to which they want to send a message. Once a member, they must supply a integer corresponding to a Guild Id. One way is to extract a Guild **`id`** from a selected Guild object **`myGuild`** and use this to send a chat message:

```
ZonaGuildChatMsg* msg =
       new ZonaGuildChatMsg(guildId, recipient_entityId,
    subject, strlen(subject), message, strlen(message));
m_zona->getGameGuildServices()->sendChatMessage(msg);
```

The Guild Chat Message has been sent by the Client. Following server-side validation, it will be broadcast to all Clients with membership of the specified Guild and received by them using callback Chat handlers.

You can derive a Guild id from its name using the **`ZonaServices::getIdForGuildName()`** function.

Receiving Chat

Clients receive incoming Sphere Chat Messages by registering a chat handler and creating a chat callback to receive data from the Terazona servers. To receive the Chat Message body data, you must extend **`ChatCallBack::onReceiveChat(ChatMsg* msg)`** and register the derived class with ZonaServices to receive Chat updates.

You should immediately copy the **`msg`** body into your own local storage because the callbacks are scoped within the Client-side Entity Object and the CAPI does periodic, scoped cleanup of old **`msg`** objects.

You must additionally set a chat handler within ZonaServices to process this callback event. There are three specific chat handlers:

Table 16-5. ZonaServices - Sphere Chat Handlers

ZonaServices Function	Priority	Description
`setChatHandler (int entityId, ChatCallback &chatCallback)`	Highest	Associate a Chat Message Handler with a specific Entity or Guild.
`setChatHandler (ChatType type, ChatCallback& chatCallback)`	Medium	Associate a Chat Message Handler for a specific **`ChatType`**.
`setDefaultChatHandler (ChatCallback &chatCallback)`	Lowest	Associate a default Chat Message Handler.

The Priority setting indicates that if a Client receives a message that can be processed by a variety of chat handlers, the message will be processed with the following precedence:

1. Handler for specific Guild / Character.
2. Handler for a specific type of message.
3. The Default Handler.

The code invocation to set a chat handler looks like this:

```
m_zona->setDefaultChatHandler(*msg);
m_zona->setChatHandler(48087, *msg);
```

Stopping Sphere Chat Monitoring

When a Client no longer wishes to receive incoming chat messages, there are corresponding ZonaServices functions available to cease monitoring specific Chat Handlers. This enables you to make available to players "ignore" or "mute" functions. The functions are:

Table 16-6. ZonaServices - Stop Monitoring Sphere Chat Functions

ZonaServices Function	Description
`stopMonitorChat (int entityId)`	Unsubscribe the Chat Message Handler associated with the specified Entity or Guild.
`stopMonitorChat (ChatType type)`	Unsubscribe the Chat Message Handler for a specific `ChatType`.
`stopMonitorChatDefault (void)`	Unsubscribe the default Chat Message Handler.

Guild Chat uses a different monitoring framework.

Managing Guild Objects

Working with Guilds provides a rich community experience to MMOG players requires an array of specialized Guild management functions. Guild Management is distributed in the following classes:

Table 16-7. Guild Management Classes

Class	Description
`ZonaServices`	Contains Client Guild management functions for adding, removing Guild members, joining Guilds, sending Guild messages, Guild monitoring, and so on.
`GameGuildCallback`	Callback that receives Guild activity data from the Terazona Servers.
`ZonaModelerGuild Object`	Defines the Guild object, enumerates mnemonic Guild properties, and contains serialization streaming functions for handling Guild data.
`GameGuildServiceInterface`	Contains functions used for Guild mesaging and management.

Using the Guild Object

You instantiate a local **`ZonaGameGuild`** object that incorporates specific characteristics defined within the Zona Model file using the ZMUI and all getter and setter functions to access Guild data and attributes are generated by Zona Modeler during its compile phase.

Zona Modeler enables you to define flexible level or depth of associations between Entities that can describe virtually any required game design for Character Guilds or memberships.

Within every Guild object, there are several Guild Attribute values:

Table 16-8. Enumerated Guild Attributes

Attribute	Description
`isPersistentMembership`	Membership information is persisted to database.
`isPreloadMemberships`	The memberships load as soon as the guild load. Should be turned only for guilds that are expected to have small memberships.
`isPersistentMessage`	Guild messages are persisted to database.
`isMemberInvite`	Only members can invite.
`notifyMembershipUpdate`	Membership updates are notified to all Guild members.
`membersCanPostMsg`	Members can post messages without Moderator approval.
`isSelfInvite`	Anyone can join the guild.

These (and other Guild attributes and data) are accessed using getter/setter functions auto-generated by Zona Modeler following Guild object definition within the Zona Model files. These are all the generated getter/setter functions for each developer-defined Guild objects:

Table 16-9. Autogenerated Getter Setter Functions For Guild Objects

Function	Notes
`getGuildId()`	
`setGuildId(int guildId)`	This is provided for reference purposes and should not be used by end developers.
`getGuildName (int& length)`	
`setGuildName (char* guildName, int length)`	
`getIsPersistentMembership ()`	
`setIsPersistentMembership (bool isPersistentMembership)`	
`getIsPersistentMessage ()`	
`setIsPersistentMessage (bool isPersistentMessage)`	
`getMembersCanPostMsg ()`	
`setMembersCanPostMsg (bool membersCanPostMsg)`	
`getNotifyMembershipUpdate ()`	
`setNotifyMembershipUpdate (bool notifyMembershipUpdate)`	
`getIsMemberInvite ()`	
`setIsMemberInvite (bool isMemberInvite)`	
`getIsSelfInvite ()`	
`setIsSelfInvite (bool isSelfInvite)`	
`getIsPreloadMemberships ()`	
`setIsPreloadMemberships (bool isPreloadMemberships)`	

Managing Guilds

Guilds are mainly managed through calls on **`ZonaServices`** objects using the **`GameGuildServiceInterface`** and **`GameGuildCallback`**.

Obtaining a Character's Guild Memberships

Clients call this function to receive a list of Guilds that their active Character has joined or can access:

```
GameGuildServiceInterface::fetchMemberGuilds
           (int entityId, ZonaClientGuildPtrVector& gpv)
```

Creating a Guild

After you have created a Guild object and populated it with the required attribute data, you use ZonaServices functions to send Guild management requests across the wire to the GGS. The steps to create a server-side Guild for in-game propagation are:

1 Instantiate a ZonaServices object and get the Game Guild Service Interface object.

```
m_zona = new ZonaServices(true, true);
m_zona->getGameGuildServices();
```

2 Create and populate a local Guild object with attribute data, including name and membership settings.

```
SampleGuildClient* aGuild = new SampleGuildClient();
aGuild->setGuildName(gname, gname_len);
aGuild->setIsPersistentMembership
                            (cg.isMembershipPersistable);
aGuild->setIsPersistentMessage(cg.isMessagePersistable);
aGuild->setMembersCanPostMsg(cg.membersCanPostMsg);
aGuild->setNotifyMembershipUpdate
                             (cg.notifyMembershipUpdate);
aGuild->setIsPreloadMemberships(cg.isPreloadMembership);
aGuild->setIsMemberInvite(cg.isMemberInvite);
aGuild->setIsSelfInvite(cg.isSelfInvite);
```

3 Publish this Guild object to the GGS:

```
m_zona->
  getGameGuildServices()->createGuild(aGuild, entityId);
```

Using Server-side developer-created validation code, the Terazona servers check that this Character has the necessary permissions to create this Guild and returns success or failure to the Client. If successfully created, the Guild object's public data attribute **`id`** is set to the Guild Id (which is the same type as an Entity Id).

The calling Client Character is automatically a Guild member and optionally the Guild Moderator.

Deleting a Guild

Deleting a Guild requires successful Server-side permission validation for success (that is, closed Guilds can only be deleted by the last Moderator). Clients actively monitoring the Guild will receive a notification of the Guild deletion through **`GameGuildCallback`**.

Managing Guild Activity

Clients receive incoming Guild Activities updates by registering a **`ZonaServices`** Guild update handler, and extending **`GuildCallback`** functions to receive Guild update data from the Terazona servers. Guild activities are directed using Client functions in **`ZonaServices`**, while clients receive notifications of the results of their Guild activities using the **`GuildCallback`** functions.

Ignoring Guild Activity

The function to unregister for Guild updates is:

```
m_zona->
      getGameGuildServices()->stopMonitorGuildMessage();
```

Managing Guild Membership

Sending Guild Invitations

The calling Character must have the necessary permissions to invite other Characters (with an Entity Id corresponding to the **`charId`**) to join the Guild. This is only an invitation; the receiving Character must call join the Guild to complete the joining process.

Receiving Guild Invitations

The Character specified by **`charId`** receives a notification of their invitation.

Joining Guilds

If the specified Character decides to join the Guild for which they received an invite, then the Inviter does not receive a specific callback notification if an invited member joins a Guild. They will receive the same callback as all other Guild members.

Leaving Guilds

Active Characters can decide to leave a Guild, or Moderators can also "kick" other Characters out of a Guild. The Terazona servers will perform the necessary validation to ensure that Moderators have the necessary permissions. Moderators cannot kick other Moderators out of a Guild (or remove their Moderator status which must be voluntarily relinquished).

Receiving Guild Membership Updates

All Clients can register to receive Guild membership updates. Developers must ensure that Clients synchronize with Server-side Guild membership update notifications by manually updating their local Guild objects to maintain correct Guild membership list.

Receiving Guild Moderator Data

All Characters can obtain a list of Moderators for a specific Guild, and can also receive Guild Moderator updates during gameplay.

Moderating Guilds

Active Characters who are Guild Moderators can access extra Moderator-specific functionality. Server-side validation will invalidate requests from non-Moderators for access to these functions.

Receiving Guild Message Data

There are many Guild management functions but receiving Guild message content is relatively simple and uses `GameGuildCallback` functions.

Demonstrating Guilds

This code exmaple demonstrates how to use some of the Guild API functions:

```
void getGuildMemberInfos(ZonaClientGuild* guild)
{
    ZonaBaseGuildMembershipMap* members =
                                        guild->getMemberMap();
    int numMembers = members->size();
    if (numMembers == 0) {
        return;  // the guild has no member
    }
    char **keys = new char*[numMembers];
    if (keys == NULL) {
        return; // insufficient memory
    }
    members->keySet(keys, &numMembers);
    for (int i = 0; i < numMembers; i++) {
        // keys are
        int memberId = atoi(keys[i]);
        ZonaBaseGuildMembership *member =
                                    guild->getMember(memberId);
        if (member->getMemberId()) {
            // get an entity id of the member
        }
        if (member->getGuildId()) {
            // the guild id that this member belongs to
        }
        if (member->getIsModerator()) {
            // this member is a moderator
        }
        if (member->isActive()) {
            // this member is on-line now
        }
    }
    if (keys) {
        delete [] keys;
    }
}
```

Managing Persistent Messages

The Persistent Messaging Guild subsystem uses a combination of Guild activities and Game Database updates to provide in-game email-like functionality.

Fetching Persistent Messages

Terazona provides a special convenience function that enables email-like functionality for members of Guilds with Persistent Messages and Persistent Membership attributes. Guild members can call this function after login to receive Guild messages that were sent while they were logged out or not receiving Guild messages:

```
GameGuildServiceInterface::fetchChatMessages
              (int guildId, INT64 minDate, INT64 maxDate,
                     ZonaGuildChatMsgPtrVector& msgs)=0;
```

Deleting Persistent Messages

Terazona provides a special convenience function to delete Persistent Messages. The function is:

```
ZonaServices::deleteChatMessage
                       (ZonaGuildChatMsgPtrVector msgs)=0;
```

Filtering Chat

Terazona supports content filtering using a word filter file that performs literal substitution for specified letter sequences, words, and sentence fragments. Filtering can be done within the GGS, or within Clients, or both. Each filtering mode has both advantages and disadvantages.

You can control whether the Auditing Server audits Chat messages in pre- or post-filter version using the **`Chat filtered="true|false"`** configuration setting in **`zona.xml`**. See *Configuring the Auditing Server on page 66* of the *Terazona Installation and Configuration Guide* for details.

Understanding Chat Filtering

With Server-side filtering, the word filter file is installed on the GGS. Server-side filtering has a run-time performance penalty because every Chat message payload is checked against the list of Server-defined substitutions by the GGS. Server-side filtering is protected against hacking and snooping, and additionally can reduce outbound Server->Client bandwidth slightly by eliminating or reducing undesired chat message payload before sending it to Clients.

With Client-side filtering, a selected portion of the word filter file is pushed to Clients during the login process and maintained by the Client during that login sessions. Client-side filtering distributes the CPU filtering burden among all Clients but the initial download does add lag time to the login process. One benefit is that Clients-side content filtering is symmetrical: Clients filter both outgoing and incoming Chat messages.

Filtering the outgoing Client->Server message content can reduce both Server bandwidth (by potentially eliminating or reducing undesired chat message payload before it ever reaches the Server) and Server load (by potentially reducing the quantity of messages requiring substitution).

Server-side content filtering is potentially very CPU-intensive, especially for large lists of substitution directives. Exercise caution when enlarging the number of Server-side content filter directives.

Writing Chat Filter Directives

The word filter file is parsed from the beginning to the end of the file and uses a simple syntax with three operators:

Table 16-10. Word Filter File Operators

Operator	Description
`[server]`	Begins the list of words to be Server-side substituted. There are no modifiers. The GGS will immediately proceed to the next line of the file.
Unicode character(s)	One or more Unicode characters. These form either the input or the output string. Strings do not have to terminated. White space does not have to be quoted or delimited.
`=`	The substitution operator. Directs the GGS to finish parsing the input string and begin parsing the output string.
`[client]`	Begins the list of words to be Client-side substituted. There are no modifiers. The GGS will immediately proceed to the next line of the file and push all substitution directives following the `[client]` operator to individual Clients during their login.
newline	Newlines (either CR or CR/LF) after the output string indicate that this substitution directive is complete.

The word filter file is deployed on the GGS in this location:

```
%ZONA_HOME%\config\zona_word_filter.ini
```

The syntax of a Chat Content Filter Directive is:

```
input string = output string
```

- ***`input string`*** indicates the input string to be substituted.
- = is the substitution operator
- ***`output string`*** indicates the output, replacement string

You can perform elimination instead of substitution by placing no characters after the = operator, using this syntax:

```
fnord =
```

This will completely block "fnord" from being seen by Clients.

This is a sample **zona_word_filter.ini** file:

```
[server]
kill=love
[client]
adam lang=slickdaddy
fnord=
```

Enabling Client-Side Filtering

Client-Side Filtering is automatically activated when there are one or more substitution directives following the **[client]** operator. This content is pushed to Clients during their login process. To enable Client-side content filtering, edit this file:

```
%ZONA_HOME%\config\zona_word_filter.ini
```

Client-side content filter directives will be pushed to Clients during their next login.

Disabling Client-Side Filtering

To deactivate Client-side filtering, remove any directives below the **[client]** operator from this file:

```
%ZONA_HOME%\config\zona_word_filter.ini
```

No Client-side content filter directives will be pushed to Clients during their next login.

Programming Client-Side Filtering

You can activate and de-activate Client-Side filtering during program execution using the **ZonaServices.setFilterOptions(TRUE|FALSE)** function.

Demonstrating Chat

Terazona's Chat architecture is feature-rich. You should examine the Win32 application, ZChatter, to see how many of these features are used in practice:

```
%ZONA_HOME%\samples\ZChatter
```

Chapter

17

Server Chat API

The Chat Plugin API (CHATAPI) enables game developers to validate and filter inter-player communication within Terazona. You can write your own validation routines and logic for guild chat messages in a similar manner to how you write code using the GSAPI to validate game state messages and inter-GSS state data transfer.

Understanding Chat Validation

The GGS Chat Plugin is a C dynamic link library (DLL) loaded by the GGS and uses a model similar to that used by the GSS Plugin. The GGS receives CAPI calls, validates them against its logic and constraints, and then sends information to the relevant Clients using CAPI chat callback functions.

The key file for implementing the CHATAPI is **`ZonaGameGuildValidate.h`**. This header file contains all of the calls used by the GGS to handle Chat validation as virtual functions that you must implement. This file is located here:

```
%ZONA_HOME%\include\ZonaGameGuildValidate.h
```

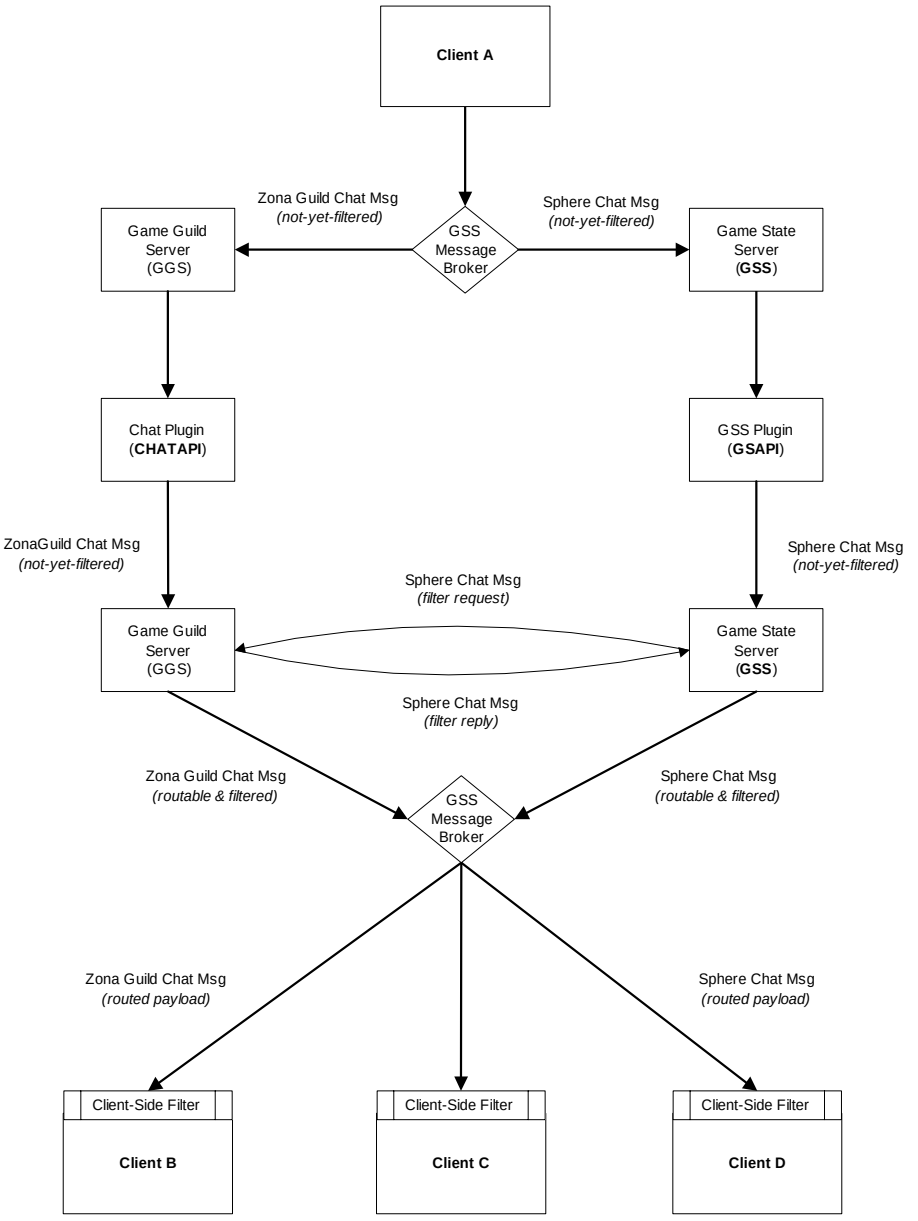
Chat Validation centers around validating Client-initiated Guild management activities. Guilds are always one of the core features of a successful MMOG and many players form deep attachment to "their" Guilds. Ensuring that the Guild system performs securely and maintains system integrity is crucial to customer satisfaction. An arbitrary or illogical Guild system will produce discontented players.

You implement the functions in **`ZonaGameGuildValidate.h`** and use them to create a robust Guild Activities validation system for your game.

Understanding Chat Message Flow

The following diagram illustrates the flow of chat messages in Terazona:

Figure 17-1. Terazona Network Message Flow



A Client transmits either a Guild Chat message or a Sphere Chat message. The message is routed to the GSS Message Broker (GSSMB) associated with the GSS responsible for managing this Client. The GSSMB routes Guild Chat messages to the GGS for processing, while Sphere Chat messages get routed to the Client's GSS.

For Guild Chat messages (distributed according to Character Ids, Guild memberships, and message flags) the CS validates requested Guild activities and message distribution using developer-defined CHATAPI functions. If configured to do so, the CS also filters the Guild Chat messages (a type of Guild object) before sending them back to the GSSMB for routing.

For Sphere Chat messages (distributed according to Character location, Region coverage, and Character preferences) the GSS processes the Sphere Chat messages using developer-defined GSAPI functions. It then requests that the CS perform content filtering (if the CS has been so configured) before sending the Sphere Chat messages (a type of Game State Message) back to the GSSMB for routing.

The GSSMB attaches routing information to the messages and hands them off to the Terazona messaging layer. This subsystem then forwards the messages to the specified Clients.

Implementing the CHATAPI Plugin

The Chat Plugin should be implemented as a DLL that interfaces with the server through system defined functions in the **`ZonaServerPlugin.lib`** static library. The name of the resulting DLL should be suitably configured in the **`zona.xml`** configuration file. As well as the GGS, the Game State Servers (GSSs) and the Sphere Server use this file to load their appropriate libraries at startup. When compiled, the same DLL is installed on all Terazona servers. The DLL exposes different functionality at runtime depending on which server it's running on.

For example, the TileTest demo application provided with the release uses a plugin DLL called **`TileTest_ServerPlugIn.dll`**.

The corresponding entry in the **`zona.xml`** file is:

```
...
<ClusterCommon>
    <LibName authDynamicLibName="AuthSamplePlugin.dll"
                        cryptoDynamicLibName="ZonaCrypto"

processDynamicLibName="TileTest_ServerPlugIn"/>
...
</ClusterCommon>
```

Compiling the CHATAPI Plugin

For details of how to compile and test the CHATAPI Plugin, please see *Compiling the GSAPI Plugin and CAPI Executable on page 110* and *Debugging the GSAPI Plugin on page 112*.

Managing Chat Validation

Client-initiated Guild management calls trigger validation routines in the GGS. The results of this validation are sent back to the calling Client and to other subscribed Clients by the GuildCallback mechanism (see *Managing Guild Membership on page 81*).

Analyzing the Chat Validation Functions

There are many Chat validation functions (all related to Guild activity):

```
bool  onValidateCreateGameGuild
                  (ZonaServerGuild* guild, int entityId)
```

Called when an entity attempts to create a game guild.

```
bool  onValidateDeleteGameGuild
                  (ZonaServerGuild* guild, int entityId)
```

Called when an entity attempts to delete a game guild.

```
bool  onValidateLoadGameGuild (ZonaServerGuild* guild)
```

Called when a guild is loaded into the GSS.

```
bool  onValidateInviteMember
  (ZonaServerGuild* guild, int inviterId, int inviteeId)
```

Called when a member tries to invite another entity to join the guild.

```
bool  onValidateKickMember (ZonaServerGuild *guild, int
memberId, int targetId)
```

Called when a member tries to kick another member out of the guild.

```
bool  onValidateAddGameGuildModerator
 (ZonaServerGuild* guild, int memberId, int moderatorId)
```

Called when a member attempts to add a moderator to the guild.

```
bool  onValidateRemoveGameGuildModerator
               (ZonaServerGuild* guild, int moderatorId)
```

Called when a member attempts to remove a moderator from the guild.

```
bool  onValidateJoinGameGuild
                 (ZonaServerGuild* guild, int entityId)
```

Called when an entity attempts to join a guild.

```
bool  onValidateLeaveGameGuild
                 (ZonaServerGuild* guild, int entityId)
```

Called when a member attempts to leave a guild.

```
bool  onValidateEnterGameGuild
                 (ZonaServerGuild* guild, int entityId)
```

Called when a member attempts to enter a guild.

```
bool  onValidateExitGameGuild
                 (ZonaServerGuild* guild, int entityId)
```

Called when a member attempts to exits a guild.

```
bool  onValidateGameGuildGameState
                  (ZonaServerGuild* guild, int entityId,
                       char* stateData, int stateLength)
```

Called when a member sends a game state to the guild.

```
bool  onValidateGameGuildChat (ZonaServerGuild *guild,
int entityId, char *chatSubject, int subjectLength, char
*chatBody, int bodyLength)
```

Called when a member sends a chat message to the guild.

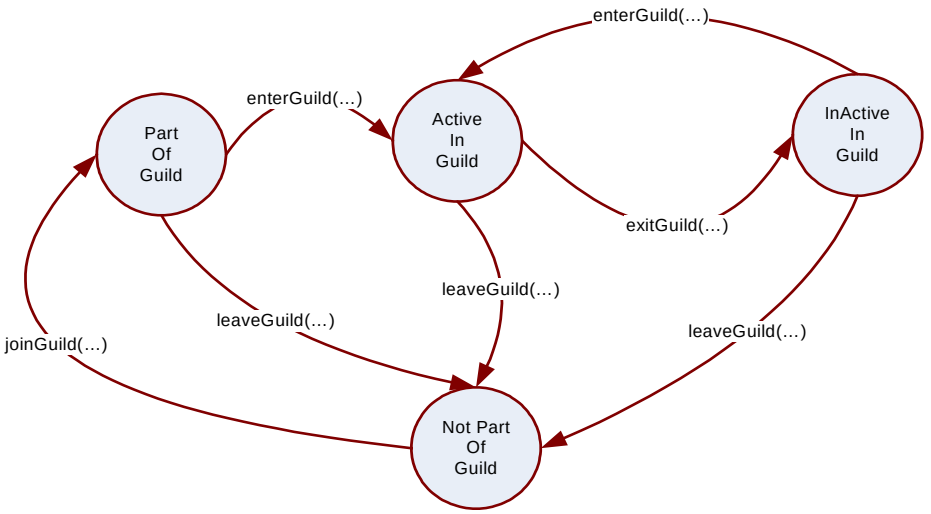
```
bool  onValidateGameGuildEntityPropertyUpdate
      (ZonaServerGuild* guild, ZonaServerEntity* entity,
                          ZonaServerEntity* previousEntity)
```

Called when a member sends an entity Update to the guild.

Understanding the Game Guild State Process

This is the Game Guild State Process:

Figure 17-2. Game Guild State Process



State – Part of Guild

The Membership object created and the Member is allowed to enter Guild.

State – Active in Guild

All Members can send and receive Guild messages.

State – Inactive in Guild

Members can no longer can send or receive messages.

State – Not Part of Guild

The associated Membership object gets deleted.

Validating Guild Creation

The **`ZonaGameGuildValidate::onValidateCreateGameGuild()`** function is called in response to a Client **`GameGuildServiceInterface::createGuild()`** function call.

Within your implementation of the **`onValidateCreateGameGuild()`** function you can check whether the calling active Character has sufficient permissions and eligibility to create the requested type of Guild.

There is no direct **`GameGuildCallback`** function but the calling Client will receive a return value indicating success or failure at creation.

Filtering Server-Side Chat

As described in *Filtering Chat on page 84*, server-side chat content filtering is possible but has significant performance characteristics. For configuration and syntax instructions, please see *Writing Chat Filter Directives on page 86*.

Auditing Chat

Auditing Chat requires no coding but is controlled through a combination of configuration and database management. The GGS writes its Chat audit data directly into the Audit Database with little intervention from a development viewpoint. For further details on auditing chat, please see *Auditing Chat Messages on page 69* and *Auditing Guild Activities on page 69* of the *Terazona Installation and Configuration Guide*.

You can control whether the Auditing Server audits Chat messages in pre- or post-filter version using the **`Chat filtered="true|false"`** configuration setting in **`zona.xml`**. See *Configuring the Auditing Server on page 66* of the *Terazona Installation and Configuration Guide* for details.

Chapter

18

Failover & Fault Tolerance

This chapter explains how Terazona transparently handles failover and creates a fault-tolerant environment.

Making Fault Tolerance and Failover Transparent

The Terazona system provides a transparent fault tolerant failover mechanism between the Game State Servers in order to allow for continuous game play. Developers need only be concerned with the failover from a conceptual level - Terazona invisibly handles the required infrastructure.

Upon the failure of a GSS (or a message broker) to which a Client was connected, each of the affected game Clients, including the NPC Server, will reconnect to a different GSS. The reconnect process is similar to the login process, where the Clients first request the Dispatcher for the least-loaded GSS. The Clients will then reconnect to this new GSS. The new GSS first attempts to retrieve a replica of the Client's state information. If this replica is unavailable, then the new GSS retrieves the Client state from the database.

As well as this reconnection, the Sphere Server transfers all the failed GSS's Region Ownerships to other available GSSs. No special coding is required by the developer to handle server failures.

Chapter 19

Cheat Prevention

This chapter describes how Terazona enables developers to prevent cheating with MMOGs.

Configuring the Cheat Prevention Interface

Terazona also provides a complete cheat prevention interface. This is done through the validation calls on the GSAPI side. The user may add new validation code to the GSAPI on the GSS and start up the new GSS within the cluster in order to add the new validation code. We do have some of the more common cheats already validated directly by the GSS.

The cheat prevention parameters can be set in the **`zona.xml`** file under the **`ZONA_HOME\config`** entry:

```
ZonaProperties
  Clusters
    Cluster
      GssServers
        GssServersCommon
          <CheatPrevention gsRateCheckFreq="10"
              gsUpdateQueSize="100"
              maxGSUpdateRate="0"/>
```

So under the above configuration based on **`maxGSUpdateRate`** a user can send in 15 game state updates per second. If this value is set to 0 then cheat detection is turned off.

- **`gsUpdateQueSize`** is number of messages the average updates per second is calculated
- **`gsRateCheckFreq`** is the frequency at which this check is made that is, a value of 10 will check one out of every ten game state packets.

Chapter

20

Game State Records

This chapter describes how Terazona enables rapid yet validated communication between game objects using a special memory structure called a Game State Record (GSR).

Analyzing Game State Records

Game State Records form the interface between the GSAPI and the GSS. They utilize an efficient shared memory model to rapidly exchange data between the GSS and the developer's GSAPI plugin.

The GSS and the GSAPI exchange information with the help of game state records (GSR). A GSR is a structure mapped into shared memory, which is accessed by both the GSS and the GSAPI library. This mechanism avoids costly memory allocations and memory copies.

One complication of this technique is that Clients must use the unstructured message publishing calls, such as **`sendGameStateMsg()`** and **`onReceivedGameStateMsg()`**.

The server passes the game state to the GSAPI in the form of a GSR. The GSAPI can access the most recent GSR with the help of the **`getCurrentGameStateRecord()`** function. This record is updated with changes in Region information for the entity in question.

If new entity state records need to be published, a call to **`getNewGameStateRecord()`** returns a reference to a new Game State Record mapped in shared memory. The structure is then populated accordingly.

The GSR structure is described as follows:

```
struct GameStateRecord
{
    int recordType; // This value should never be modified
    int entityId; // The entity id for corresponding
                  // to state update
    int entityRegionId; // The Region id of the above entity
    int stateDataOffset; // The offset for
                         //  reading & writing state data
                       // This value should never be modified
    int operation; // The operation (bit combinations)
                    // for this state
    int stateDataLen; // The length of the state data
                      //to be published
// This value should not be modified if it has been
// obtained by a call to getCurrentStateRecord
}
```

The function descriptions are below:

1 **`GameStateRecord* getCurrentGameStateRecord()`**

- Return: reference to the most recent game state record sent by the Client
- Returns a reference to the most recent GSR. The GSR can be populated with updates to the entity's Region information.

A GSR obtained by a call to **`getCurrentGameStateRecord()`** should only have modifications to the **`entityRegionId`**, **`RegionCoverage`** and **`operation`** fields. The rest of the fields are cross-referenced by the Server in the course of its processing and should not be modified.

2 **`GameStateRecord* createNewGameStateRecord (short length, byte* entityDataPtr)`**

- Return: reference to a new GSR.
- Returns a reference to new empty GSR. The GSR can be populated appropriately.
- The **`length`** field indicates the size of the gamestate to be published.
- The **`entityDataPtr`** reference is set to point to the first byte of the free memory block of size indicated by length.
- This function is mostly called when the GSAPI has to generate and publish new state information to the clients.
- A new flag called **`PUB_OP_CLIENT_PRIVATE`** has been introduced in the *operation* field to indicate **`publishGameStateTo`** functionality.
- The only difference between the two functions is, that the operation field for the **`publishGameStateTo`** function is Bitwise OR'd with the new **`PUB_OP_DIRECT`** flag.

Upon returning from any JNI call, the Server flushes the shared memory and all references to it are invalidated. Therefore the memory is valid only in the context of that particular function call.

Part

III

Developing With Terazona

This part of the Terazona Developer Guide analyzes some of the client-server interactions within Terazona and demonstrates how to develop some simple applications.

Chapter

21

Development Environment

This chapter describes how to set up a typical Terazona development environment.

Compiling the GSAPI Plugin and CAPI Executable

The CAPI can be built as an executable **`.EXE`**.

The GSAPI Plugin must be built as a dynamic library. It must be linked with the **`ZonaServerPlugin.lib`** static library. The dynamic library uses the functions defined within the **`ZonaServerPlugin.lib`** library to interface with the GSS.

There are various header file containing the function declarations, which have to be implemented by the developer:

```
ZonaBaseEntity.h
ZonaCharacterValidate.h
ZonaChatGuildValidate.h
ZonaEntityValidate.h
ZonaGameGuildValidate.h
ZonaGameStateValidate.h
ZonaRegionValidate.h
ZonaServerCharacter.h
ZonaServerEntity.h
ZonaSystem.h
ZonaTimerEvents.h
```

- **`ZonaGSPublish.h`** is a header file containing the various functions, declarations, Entity state updates, and Timer functions. These are pre-defined and do not require implementation.

To compile the GSAPI, using Microsoft Visual C++:

Include the header(.h) files from:

```
%ZONA_HOME%\include
```

For both Client executables (CAPI) and Server Plugins (GSAPI), ensure you also include the header(.h) files from:

```
%ZONA_HOME%\include\ZonaModeler
```

Project -> Settings... -> C++ -> Category: Preprocessor

Under “Additional Include Directories” add:

```
%ZONA_HOME%\include
%ZONA_HOME%\include\ZonaModeler
```

Project -> Settings... -> C++ ->Category: Code Generation

Ensure the runtime library for debug is “debug multithreaded” and for release is “multithreaded”

Project -> Settings... -> Link->Category:Input

Under Object->Library Modules, you need to enter Terazona-specific targets for linking:

```
All Configurations:
$(ZONA_HOME)\lib\ZonaServerPlugin.lib
```

Debugging the GSAPI Plugin

Because the GSAPI Plugin is either a Windows DLL or a shared library, you can't simply debug it from your debugger. This section introduces two ways to debug using the Windows platform. You can choose one of them to meet your need.

Using DebugBreak()

The DebugBreak function causes a breakpoint exception to occur in the current process. See the details in:

```
http://msdn.microsoft.com/library/default.asp
                  ?url=/library/en-us/debug/base/debugbreakop
```

When the exception is thrown, you will see the Window Error Report pop-up that allows you to starts VC++ and debug.

Here is the step-by-step instruction for TileTest:

1. Insert **`DebugBreak();`** before the line where you want to debug.
2. Build TileTest_ServerPlugIn in the debug mode, which will produce the debug version DLL in ZonaHome\bin.
3. Configure the server to load the debug version by changing the following entry in **`$ZonaHome\config\zona.xml`**:

- **`pluginDynamicLibName="ZonaBattle_ServerPlugIn"`**

4. Start Sphere Server or GSS as usual.
5. When you come to the point where you put **`DebugBreak()`**, you will see the Windows Error Report screen pop-up, click on the Debug button which will start VC++ with the disassembly and your C++ GSAPI code running in debugger.
6. Hit F10 to advance to the next line corresponding to the C++ source code, and switch to the C++ view window.
7. You should be able to set breakpoints, inspect variables, if needed.

Setting up VC++

You can set up VC++ to debug the GSAPI. Setting VC++ is not as simple as inserting **`DebugBreak()`**. But, once you set it up, you can start GSS from VC++ and set breakpoints in the same way as you set them into a standalone program.

Here is the step-by-step instruction for TileTest:

1. Configure the server to load the debug version by changing the following entry in **`%ZONA_HOME%\config\zona.xml`**:
   - **`pluginDynamicLibName="TileTest_ServerPlugIn"`**
2. Build TileTest_ServerPlugIn in the debug mode, which will produce the debug version DLL in **`$ZonaHome\bin`**.
3. Go to the Project->Settings->Debug window.
4. In the "Executable for debug session:", set up to invoke **`java.exe`**.

```
C:\$JAVAHOME\bin\java.exe
```

5 In the "Program arguments", supply all the info that **`java.exe`** needs to run GSS.

```
-cp
.;c:\Zona\Server1.4.1\lib\zona_server.jar;c:\Zona\Serv
er1.4.1\lib\zona_client.jar;c:\Zona\Server1.4.1\utils\
lib\imq.jar;c:\Zona\Server1.4.1\utils\lib\jndi.jar;c:\
Zona\Server1.4.1\utils\lib\jms.jar;c:\Zona\Server1.4.1
\utils\lib\fscontext.jar;c:\Zona\Server1.4.1\utils\lib
\icejms.jar;c:\Zona\Server1.4.1\utils\lib\msbase.jar;c
:\Zona\Server1.4.1\utils\lib\msutil.jar;c:\Zona\Server
1.4.1\utils\lib\mssqlserver.jar;c:\Zona\Server1.4.1\ut
ils\lib\jdbcpool-
0.99.jar;c:\Zona\Server1.4.1\utils\lib\tinySQL.jar;c:\
Zona\Server1.4.1\utils\lib\db-ojb-
1.0.rc3.jar;c:\Zona\Server1.4.1\utils\lib\commons-
pool.jar;c:\Zona\Server1.4.1\utils\lib\commons-
collections-
2.0.jar;c:\Zona\Server1.4.1\utils\lib\commons-lang-
1.0-mod.jar;c:\Zona\Server1.4.1\utils\lib\commons-
dbcp.jar;c:\Zona\Server1.4.1\utils\lib\xmlParserAPIs.j
ar;c:\Zona\Server1.4.1\utils\lib\xercesImpl.jar;c:\Zon
a\Server1.4.1\utils\lib\xml-
apis.jar;c:\Zona\Server1.4.1\utils\lib\xalan.jar;;c:\Z
ona\Server1.4.1\utils\lib\WkJavaApi.jar;c:\Zona\Server
1.4.1\ext\classes;c:\Zona\Server1.4.1\ext\ojb;c:\Zona\
Server1.4.1\ext\metadata;c:\Zona\Server1.4.1\ext\class
es -Dzona.dev= -Dzona.home=c:\Zona\Server1.4.1
```

6 Set breakpoints and debug as usual.

Testing the Client

Configuring the XML File

The XML File must be configured to point the various servers at the proper location of the all of the necessary servers as well as their configurations. The setup and configuration for Terazona is controlled by a single XML file:

```
%zona_home%\zona.xml
```

The configuration is split into two parts:

1. System Configuration
2. Cluster configuration

The System branch controls configuration values that are good for the entire system.

The Cluster branch has common areas for the clusters, and each type of server.

The Installer modifies the critical items in the **`zona.xml`** file during the install process. If you wish to change these values after install, you must edit the this file. In the next major release of Terazona. Terazona will support the Terazona Cluster Editor ("Clusternator") that will enable modification and management of XML files across the cluster from one central screen.

For now, these are the critical values in the **`zona.xml`** file that developers can modify:

```
ZonaProperties/Clusters/Cluster/ClusterCommon/Profile/
database@serverURL= database URL
```

that is,

```
jdbc:microsoft:sqlserver://
zonafile:1433;DatabaseName=ZonaDB;SelectMethod=cursor

ZonaProperties/Clusters/Cluster/ClusterCommon/Profile/
database@userName= [database username]
ZonaProperties/Clusters/Cluster/ClusterCommon/Profile/
database@password= [Database password]
```

If the Server locations change, you should use the Installer to reinstall and reconfigure that the **`zona.xml`** file.

Changing the Development Options

To develop your own applications or control which Plugin gets loaded by the Server, you should change this value:

```
pluginDynamicLibName="ZonaBattle_ServerPlugIn"
```

Change **`ZonaBattle_ServerPlugIn,`** to the name of your compiled DLL. This will change the name of the main Server plugin.

Starting the Client

For initial tests of the client, it is probably best to start without the NPC Server. This will facilitate ease of use and a simpler debugging path. Run the following servers from the Start Menu in the **Start > Programs > Terazona Server 1.4.1** location in the order below.

1. **Servers > Authentication server**
2. **Servers > Messaging server**
3. **Servers > ZAC server**
4. **Servers > Dispatcher server**
5. **Servers > Sphere server**
6. **Servers > Game State Server**
7. **Servers > Game Guild Server**

Initial tests should also start with only a single GSS to simplify the debugging process.

Chapter

22

Introducing Zona Modeler

This chapter explains how and why Zona Modeler (ZM) exists and how it integrates within Terazona. ZM facilitates the rapid creation and modification of bandwidth-optimized game objects within networked game architectures. ZM auto-generates Java and C++ code for easy Client- and Server-side integration and deployment.

Introducing Zona Modeler

Zona Modeler (ZM) is a game object design suite that provides an environment and a set of tools that enables developers to prototype, create, and edit persistable Game Objects and bandwidth-optimized game state messages. Zona Modeler enables game developers and game designers to create, modify, and compile network game objects (NGOs) using a simple XML-based definition format.

Zona Modeler is focused on the creation, modification, and persistence of network objects. Terazona games typically execute across large, possibly heterogeneous game server clusters. A key component of any such distributed system is describing an efficient method of communicating object state changes between different machines in the cluster and persisting these state changes to a database.

This process of sending object states and object state changes between execution machines across the "wire" (or network layer) within a distributed system is called marshalling (for sending) and unmarshalling (for receiving). This is similar to the concept of serialization.

Within distributed systems, to perform marshalling and unmarshalling each execution machine requires various meta-object support data commonly known as interfaces, skeletons, and stubs. Zona Modeler creates these for you during design-time.

Because ZM also provides a bi-directional mapping between Game Objects and database objects, developers can use industry-standard RDBMS to store and serve Game Objects. Game applications (and game application components) can be made database-aware and utilize database and web services to enhance interoperability and cross-platform deployment. This persistence layer supports fail-over and is fault-tolerant.

Zona Modeler's Components

In terms of component view, Zona Modeler consists of two major components.

a **Zona Modeler Design Studio**

Engage developer in designing the game object model and also audit object model for that game object model

1 ZM UI – a pure Java IDE to enable freeform game object design.

2 ZM Processor – process UI-generated game object models and generates object-relational mappings and C++ or Java sources for Clients, Servers, and Audit Servers.

3 ZM Deployer – Deploys the custom game schema, table, and game data to the Game and Audit Database, compiles the generated server components and prepares metadata base for runtime usage.

b **Zona Modeler Runtime framework**

This framework sits beneath the Zona Application Framework (ZAF) and provides runtime functionalities such as persistence, custom bandwidth optimized messaging for generated ZM game objects.

Understanding the Zona Modeler Architecture

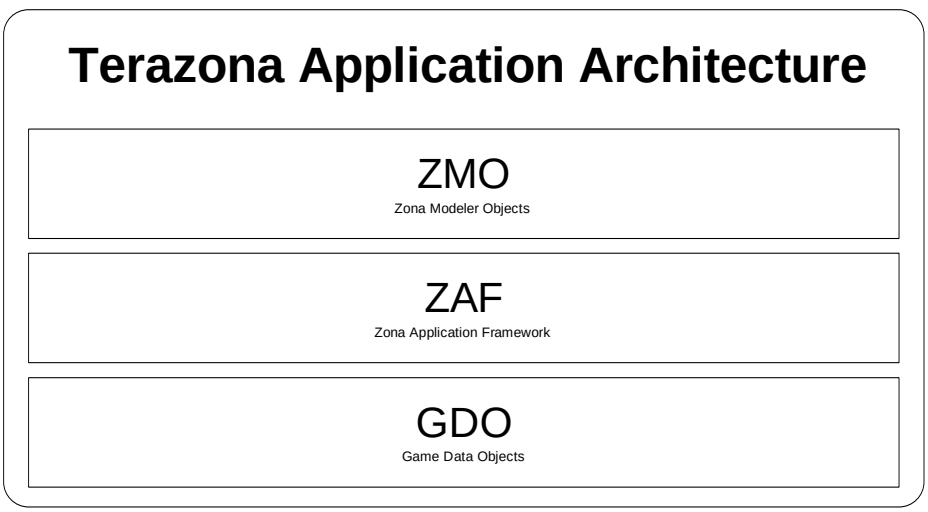
A Terazona application can now be represented by a 3-level architecture. The base level, ZMO, handles network game objects and is transparent to game developers and game designers. Terazona developers interact primarily with the middle layer, ZAF, that handles in-game object execution and management. The final layer, GDO, transparently handles in-game object instantiation.

Each data object layer talks to an associated layer-specific Controller object that provides the following services:

1. Object-Relational Mapping
2. Messaging
3. Validation

Zona Modeler abstracts the network game object layer below the Zona Application Framework (ZAF) game execution layer. To manipulate ZAF, you use the standard C++ or Java CAPI and GSAPI functions. Using the 3-layer model, the new Terazona class hierarchy looks like this:.

Figure 22-1. Terazona Application Architecture



Zona Modeler defines and manages the object substructure of a Terazona-based environment. This Zona Modeler Object (ZMO) layer sits beneath the Zona Application Framework (ZAF) layer with which you interact using the Client and Server APIs. ZM auto-generates all the run-time classes and initialization data, as well as providing object-relational persistence mapping.

At run-time, the interaction between the ZMO and the ZAF objects instantiates the Game Data Object layer, that is, the visible executing Terazona "game world". During game execution, the game object Entity state changes are persisted to the GameDB or the AuditDB.

Examining Zona Models

Zona Modeler objects are stored in an XML-based model file. Within this model there are several components:

Table 22-1. Zona Modeler Components

Attribute	Description
Model	This is the root node that with an attribute that specifies the name of the model file. In effect, this is the name of your game.
Character Entity	This element node represents Character Entities.
Child Entity	This element node represents Child Entities.
Guild Entity	This element node represents Game Guild Entities.
Config File	This is a reference to an external configuration file that contains database access parameters. It is stored as an attribute parameter value within the **`<objectmodel ... />`** element.

Within the ZMUI, you can create and edit NGOs, add or adjust their attributes, and save your progress in an XML-based model file. Every model file references a Zona Modeler configuration file that contains your schema and database access information.

When you want to compile the NGOs into in-game objects, ZMUI automates this conversion of the model file NGOs into in-game binary objects, creating database schema, server-side Java class files, static runtime support files, and C++ and Java Client-specific source code suitable for inclusion within your developed Clients.

During its compilation, Zona Modeler creates and embeds ClassIDs within the generated source code that function as persistent object references. You can define ClassIDs for certain classes of objects within the Zona Modeler UI, and these ClassIDs are persisted within the XML-based model file. The ZM code generator takes the ClassIDs from the model file and embeds them within its run-time metadata output. You can use this ClassID during run-time to reference Zone Modeler objects.

The default Terazona installation creates a single configuration file that all model files reference:

`%ZONA_HOME%\config\ZonaModelerConfig.xml` (*Windows*)
`$ZONA_HOME/config/ZonaModelerConfig.xml` (*Linux*)

You do not have to use a single modeler configuration file for all your models. If different models require access to different databases, or or you want to use different access credentials, then you can create and attach different modeler configuration files to any or all of your Zona Models.

Using Zona Modeler

There are three stages to using Zona Modeler:

Table 22-2. Zona Modeler Stages

Stage	Description
Design Time	During this stage you interactively create and modify Zona Models. When you are satisfied with your design, you compile ("Run") your model. Zona Modeler produces various output files and classes.
Deployment	During this stage you deploy the Zona Modeler output objects across your Terazona cluster. You also link some Zona Modeler output files with your Client application compiles to produce Zona Modeler-aware executables.
Run Time	During run-time the Zona Modeler-aware Client executables and the Server-side deployed Zona Modeler classes communicate transparently.

Starting Zona Modeler UI

To invoke the Zona Modeler UI, on Windows you can use **Start > All Programs > Terazona Servers *X.X.X* > Zona Modeler** icon. This invokes a Java UI within which you use Zona Modeler.

The line command to load the Zona Modeler UI (ZMUI) is

```
%ZONA_HOME%\ZonaHome\bin\startZonaModeler.bat (Windows)
$ZONA_HOME/ZonaHome/bin/startZonaModeler (Linux)
```

Understanding Zona Modeler Input and Output

Zona Modeler takes in three inputs (termed **D**, **C**, and **CA**), and produces a set of run-time metadata and classes that produce Server-, Client, and Audit-specific object instantiations during run-time.

Examining the Zona Modeler Inputs

Zona Modeler takes in three inputs. These are:

1 **Zona Model Object Definition Schema** - This is the XML-based model file created using the ZMUI. This file contains element and attribute definitions as well as some external information. This input object is referred to as **D**.

   In the demo TrackerClient, for example, this object is stored in the file:

   ```
   %ZONA_HOME%\samples\TrackerClient\TrackerSchema.xml
                                                    (Windows)
   $ZONA_HOME/samples/TrackerClient/TrackerSchema.xml
                                                  (Linux)
   ```

2 **Zona Model Configuration** - This is the XML-based environment configuration file for the Zona Modeler network data objects. This contains information such as the working directory and the database location for Zona Modeler objects (ZonaDB). Note that although they can be located within the same database server, the ZonaDB and the Game DB can be hosted on different machines. This input is referred to as **C**.

   Although it's possible to specify multiple database configurations for different Zona Modeler projects, the default Terazona installation uses a single file:

   ```
   %ZONA_HOME%\config\ZonaModelerConfig.xml (Windows)
   $ZONA_HOME/config/ZonaModelerConfig.xml (Linux)
   ```

3 **Audit Model Configuration** - This is the XML-based environment configuration file for the audited Zona Modeler network data objects. This contains information such as the working directory and the database location for audited Zona Modeler objects (AuditDB). Note that although they can be located within the same database server,

the ZonaDB, the Game DB, and the AuditDB can be hosted on different machines. This input is referred to as CA.

Examining the Zona Modeler Outputs

During its compile phase, Zona Modeler combines all three inputs to derive specific object representations suitable for deployment across Terazona. Specifically, Zona Modeler outputs C++ and Java code definitions for three Data Objects:

1. **DS** - This is the server-side view of the **D** data object. This will encapsulate the System, Public, and Private Properties of specific data object instances.
2. **DC** - This is the client-side view of the **D** data object. This will encapsulate the Public, and Private Properties of specific data object instances.
3. **DA** - This is the audit view of the **D** data object. This will encapsulate those System, Public, and Private Properties of specific data object instances designated for auditing within CA.

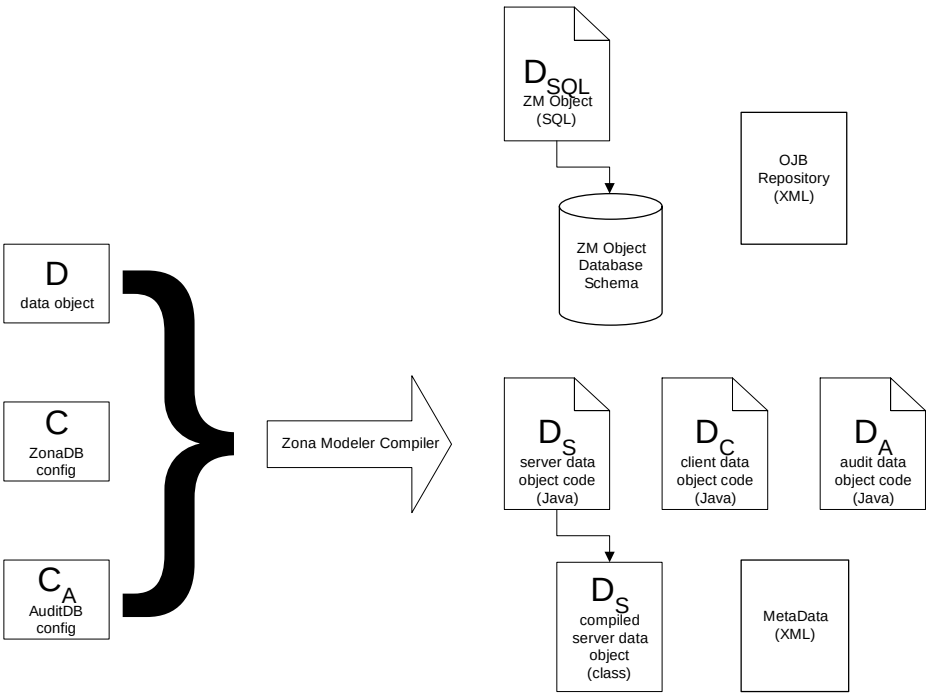
Zona Modeler also produces associated support and definition files. These are:

1. A compiled version of DS, for deployment across all Game State Servers (GSSs).
2. A SQL definition required for Terazona data object persistence. This is called DSQL.
3. A database-specific schema version of DSQL. You can chose the database within the Zona Modeler UI.
4. An XML-based MetaData file that contains the static data necessary to instantiate objects, for deployment across all GSSs.
5. An XML-based ObJectRelationalBridge (OJB) file that describes the object-relational persistence mappings that link Zona Modeler objects with specific database columns.

Zona Modeler does not compile the C++ versions of **DS**, **DC**, or **DA**. During Client development, or as desired, you can use your C++ compiler of choice to compile these objects..

6 This diagram illustrates the Zona Modeler inputs and outputs:

Figure 22-2. Zona Modeler Inputs & Outputs



Following successful execution of the Zona Modeler compile process, the following directory contains all of the ZM output files, including **DS**, **DC**, and **DA** class files and metadata information required for runtime:

```
%ZONA_HOME%\ext (Windows)
$ZONA_HOME/ext (Linux)
```

The ZM output is stored in subdirectories as follows:

Table 22-3. Zona Modeler Output Object Locations

`\ext` Subdirectories	Description
classes	Run-time server-side Java classes
metadata	Run-time object instantiation information
ojb	Run-time object-relational mapping information
src	Object source code (C++, Java).Relational table code (SQL)
debug	First-time ZM model initialization data stored as compressed archives. Useful for debugging and support.

Deploying Zona Modeler Objects

Zona Modeler deployment differs depending on whether you are deploying on the Server-side Terazona cluster, (Java classes and C++ source) or compiling into the Client-side executable (C++ source).

Deploying the Server-Side Zona Modeler Output

1. Deploy the contents of the **`\ext`** directory onto all servers within the Terazona cluster as follows:

   ```
   %ZONA_HOME%\ext (Windows)
   $ZONA_HOME/ext (Linux)
   ```

2. Ensure that the **`\ext`** directory is referenced within each server's Java **`CLASSPATH`**.
3. Build your Server plugin incorporating the generated C++ code (**`.h`** and **`.cpp`**) from:
   a. `%ZONA_HOME%\ext\src\cpp\server (Windows)`
      `$ZONA_HOME/ext/src/cpp/server (Linux)`
   b. `%ZONA_HOME%\ext\src\cpp\common (Windows)`
      `$ZONA_HOME/ext/src/cpp/common (Linux)`

Compiling the Client-Side Zona Modeler Output

1 Build your Client executable incorporating the generated C++ code (.h and .cpp) from:
   a `%ZONA_HOME%\ext\src\cpp\client (Windows)`
     `$ZONA_HOME/ext/src/cpp/client (Linux)`
   b `%ZONA_HOME%\ext\src\cpp\common (Windows)`
     `$ZONA_HOME/ext/src/cpp/common (Linux)`

During compile-time, the metadata and object-relational data is stored within your executable. As a result, these files do not require separate client-side deployment.

Using the Zona Modeler Class IDs

The Class Id created within each model is unique for each generated class and remains invariant for each Entity through subsequent ZonaModeler runs. As a result, you can use the Class Ids to downcast from **`ZonaClientEntity`** or **`ZonaServerEntity`** to your developer-defined Entities.

Refer to **`ClassId.h`** generated within **`$ProjectRoot/modeler/common`**:

```
onNotifyEntityJoinedSphere(ZonaClientEntity* zce) {
    switch ( zce->getClassId() ) {
        case …
        case ClassId_MyCharacter:
            ch = (MyCharacter*)zce;
        break;
        case …
    }
}
```

Chapter

23

Introducing the Zona Modeler UI

This chapter illustrates how to use the Zona Modeler UI (ZMUI) to graphically create and modify Characters, Child Entities, and Game Guilds into "Game" or model files. Within ZMUI you can also compile these models to generate cross-platform client and server network game objects suitable for deployment and run-time execution.

Introducing the Zona Modeler UI

The Zona Modeler UI (ZMUI) enables game developers and game designers to create, modify, and compile network game objects (NGOs) using a simple graphical interface. Within the ZMUI, you can create and edit NGOs, add or adjust their attributes, and save your progress in an XML-based model file. When you want to compile the NGOs into in-game objects, ZMUI automates this conversion of the model file NGOs into in-game binary objects, creating database schema, server-side Java class files, static runtime support files, and C++ and Java Client-specific source code suitable for inclusion within your developed Clients.

Within ZMUI, you create model files that define your NGOs. Every model file references a Zona Modeler configuration file that contains your schema and database access information. The default Terazona installation creates a single configuration file that all model files reference:

`%ZONA_HOME%\config\ZonaModelerConfig.xml` (*Windows*)
`$ZONA_HOME/config/ZonaModelerConfig.xml` (*Linux*)

You do not have to use a single modeler configuration file for all your models. If different models require access to different databases, or or you want to use different access credentials, then you can create and attach different modeler configuration files to any or all of your Zona Models.

Using the Zona Modeler UI

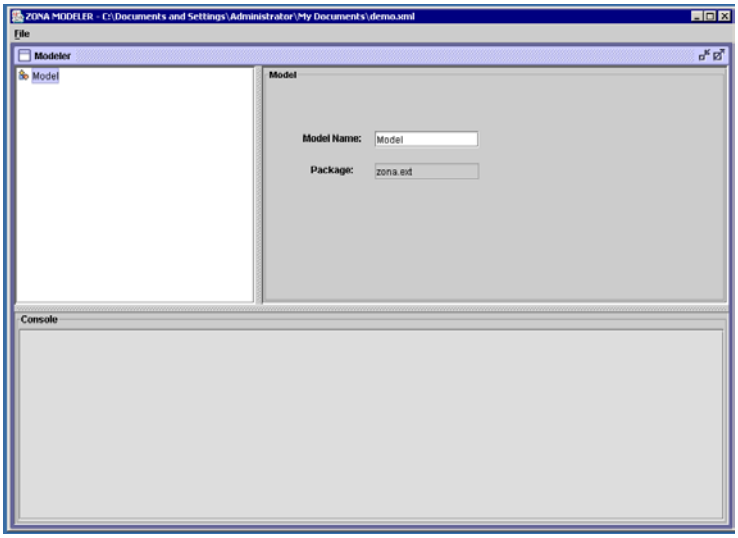
To invoke the Zona Modeler UI, on Windows you can use **Start > All Programs > Terazona Servers *X.X.X* > Zona Modeler** icon.

The line command to load the Zona Modeler UI is:

`%ZONA_HOME\bin\startZonaModeler.bat` (Windows)
`%ZONA_HOME/bin/startZonaModeler` (Linux)

When you invoke Zona Modeler, the Java-based Zona Modeler UI displays:

Figure 23-1. Zona Modeler - Initial Load Screen



These are the main ZMUI components:

Table 23-1. ZMUI Main Panel Components

Component	Description
Menu Bar	Displays on top of the UI, within the **File** object. Click the **File** object to reveal several sub-components: **File > Configure...** **File > Open** **File > Save** **File > Save As...** **File > Run** **File > Exit**
Object Tree	Displays within the top-left panel of the UI. Presents a graphical view of your model file, enabling you to traverse the element hierarchy and click specific elements for selection, modification, or deletion. The object tree always begins with a single root node with the default name **`Model`**.

Table 23-1. ZMUI Main Panel Components

Component	Description
Object Inspector	Displays within the top-right panel of the UI. The display is context-sensitive and updates to reflect whichever object you have selected within the Object Tree panel (top-left). Some properties are ghosted and non-editable because they have been auto-generated by Zona Modeler. Non-ghosted properties are editable.
Comments	Used to add user-defined text comments for selected Entities or Properties. These comments do not affect the game logic.
Console	Displays on the bottom half of the ZMUI and displays status updates, console messages, and command output.

Using the ZM UI Console

The lower panel, Console, displays the status of your activities within ZMUI, and also displays warning and error messages. All activities for a current model display within the Console and can be copied into a clipboard for reference or debugging purposes. Opening a new model or running a Zona Modeler compile clears the console and begins writing a new set or activity messages within the Console.

Using the Model File

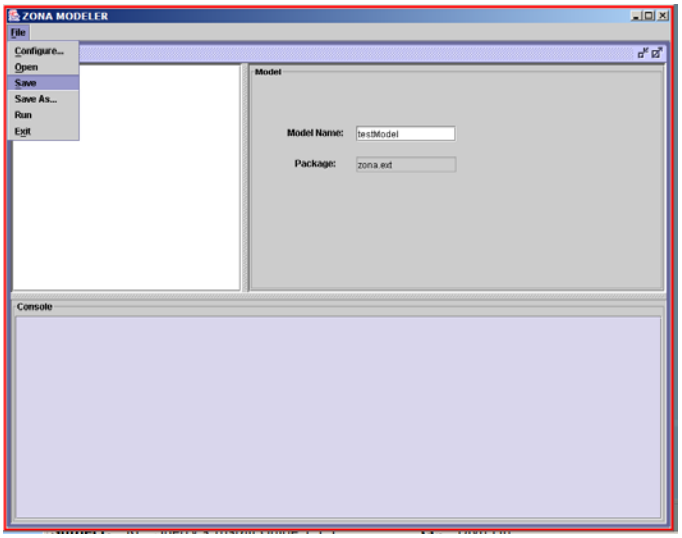
The ZMUI creates and modifies an XML-based model file that persists your changes. The Zona Modeler compiler parses this model file to produce its output.

Saving Your Model File

Save early, save often. To save your model file from within ZMUI:

1 Select **File > Save...**

Figure 23-2. Zona Modeler - Save Model Selected



2 Type a name for your on-disk model file.

3 Click the **Save** button.

4 Your change to the model name has been saved.

If your model contains only a root node, then ZMUI displays a "Model is Empty" Alert and blocks your save operation. You can only save non-empty models.

Configuring Your Model File

ZMUI initially creates a blank model file. There are two primary configuration attributes:

Table 23-2. ZMUI Primary Configuration Attributes

Attribute	Description
Model Name	This is the name of the model. You can change the name to any suitable value.
Package	This defines the namespace for the compiled object classes that will create the in-game NGOs. You cannot change the package name; it is always set to **`zona.ext`**.

There are three secondary configuration attributes:

Table 23-3. ZMUI Secondary Configuration Attributes

Attribute	Description
Current Config File	Contains a reference to the specific model configuration file (which stores all configuration data). The default Zona Modeler configuration file is: **`%ZONA_HOME%\config\ZonaModelerConfig.xml (Windows)`** **`$ZONA_HOME/config/ZonaModelerConfig.xml (Linux)`**
Project Root Dir	Contains a reference to the root directory for your IDE or development environment.
Database Configuration	Contains various database-specific access parameters and credentials. Initially defined during the install of Terazona.

There are six database configuration attributes:

Table 23-4. ZMUI Database Configuration Attributes

Attribute	Description
Platform	Specifies which database architecture to use for this connection. ZMUI displays only supported database architectures.
Host	The database server's IP address or hostname.
Port	The database server's listening IP port.
Database	The database name within the database server.
Username	A database user with authority to create Tables in the database.
Password	The authorized user's password.

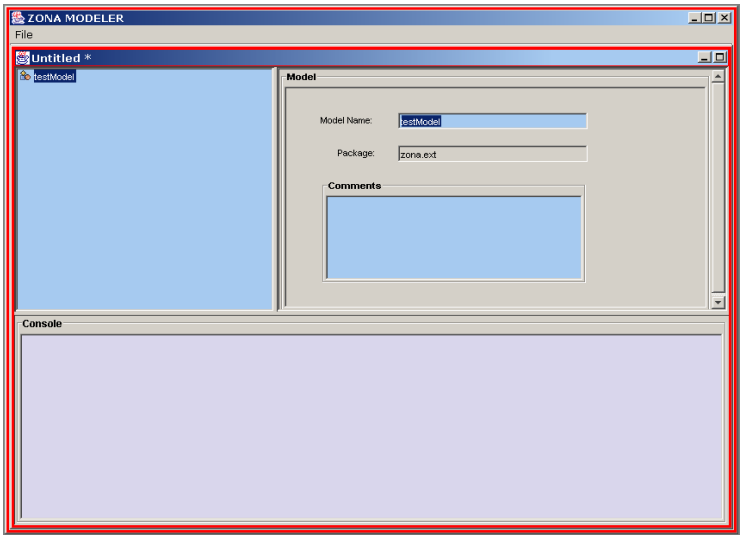
The **Audit** configuration tab with the ZMUI displays the same attributes. Use the Audit Tab to define the Audit Database Configuration.

Changing the Model File Name

To change the model file name, within ZMUI:

1 Within the right-hand **Model** panel, click in the **Model Name** text field:

Figure 23-3. Zona Modeler - Changing Model Name



2 Highlight the existing model name text.

3 Replace the existing text with your new model name.

4 Press the **Enter** key (on the keyboard) to register your change. The root node name in the left-hand panel updates to reflect your change.

5 Select **File > Save...**

6 Type a name for your on-disk model file.

7 Click the **Save** button.

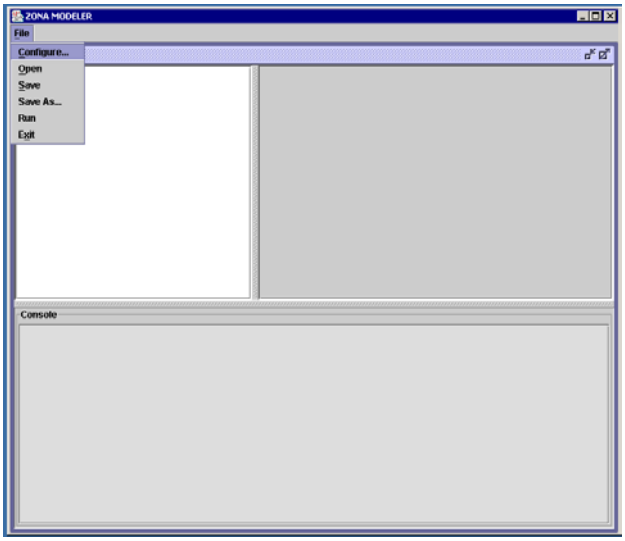
8 Your change to the model name has been saved.

Changing the Model Configuration File Reference

ZMUI remembers the last-referenced model configuration file and inserts this as the initial, referenced file for all new models. To change the configuration file, or to modify the values, within ZMUI:

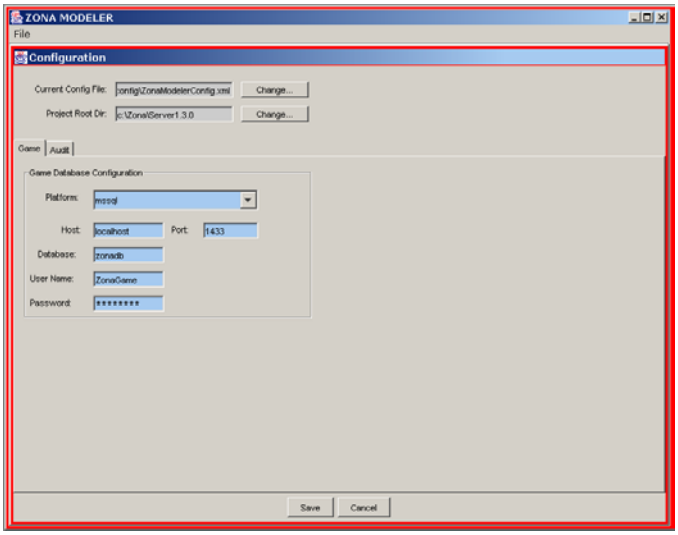
1 Select **File > Configure...**

Figure 23-4. Zona Modeler - Configuration Selected



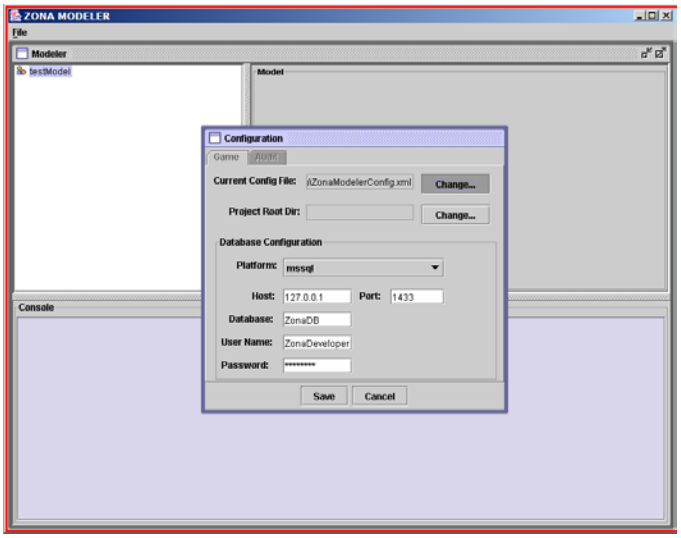
2 The Configuration dialog displays:

Figure 23-5. Zona Modeler - Configuration Display



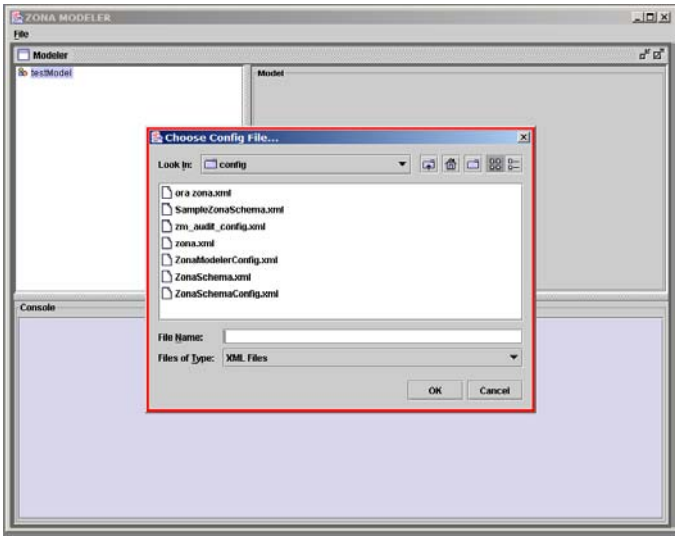
3 To change the Current Configuration File reference, click the **Change...** button on this line.

Figure 23-6. Zona Modeler - Change Current Configuration File



4 The **Choose Config File...** file browser dialog displays.

Figure 23-7. Zona Modeler - Choose Configuration File



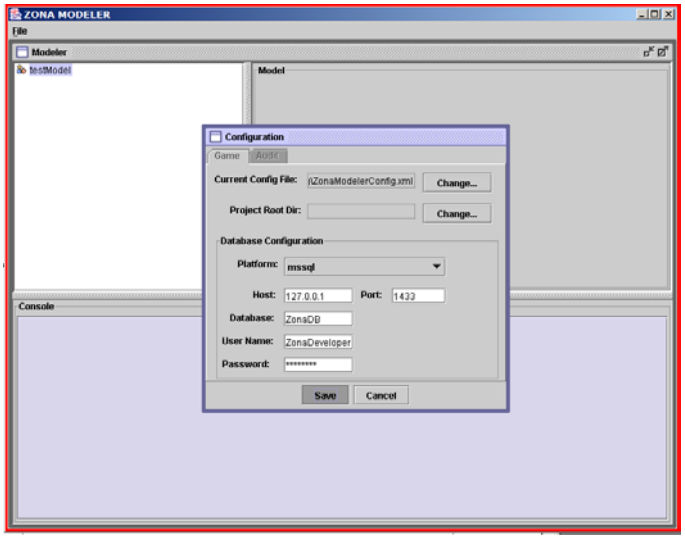
5 Type a name for your new model configuration file, or navigate to an existing model configuration file.

6 Click the **OK** button.

7 The **Choose Config File...** file browser dialog closes and the Configuration dialog again displays.

8 Click the **Save** button.

Figure 23-8. Zona Modeler - Save Configuration File Choice



9 Persist your changed model file to disk using the **File > Save...** command.

10 Your new model configuration file has been saved has been saved.

The **Audit** configuration tab displays the same configuration attributes. Use the Audit Tab to define the Audit Database Configuration.

Changing the Project Root Directory

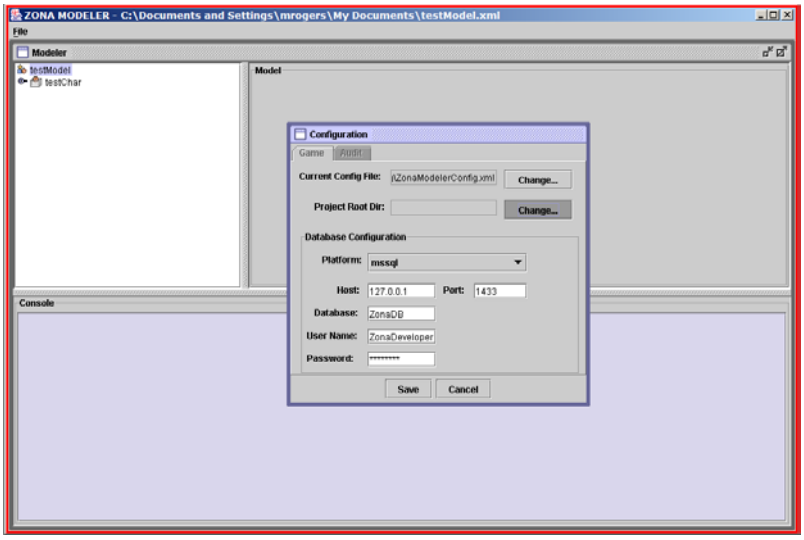
You can integrate Zona Modeler output with your development environment by specifying the location of your .DSP (or other IDE files) within the **Project Root Dir:** field. During its compile process, Zona Modeler will deploy the generated C++ source code (both `.h` and `.cpp`) files to this directory for easy integration with your other development code. To specify a project root directory:

1 Select **File > Configure...**

2 The **Configuration** dialog displays:

3 Click the **Change...** button on the same row as the **Root Project Dir.**

Figure 23-9. Zona Modeler - Change Project Root Directory

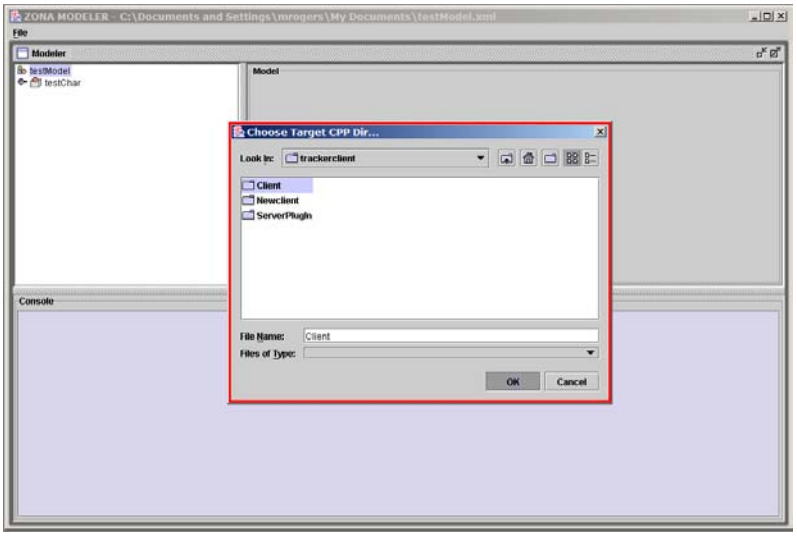


4 The **Choose Target CPP Dir...** file browser dialog displays.

5 Navigate to your selected Project directory (that contains your IDE project files).

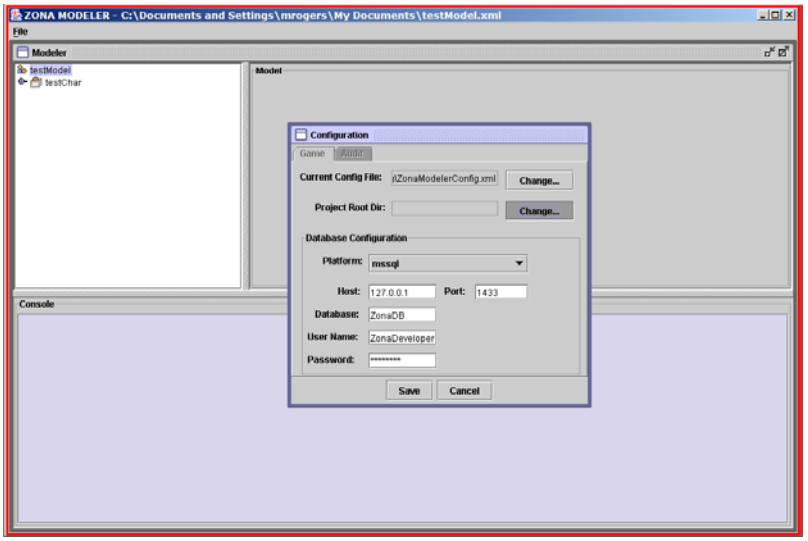
6 Highlight the selected directory and click the **OK** button.

Figure 23-10. Zona Modeler - Selecting the Project Root Directory



7 The **Configuration** dialog redisplays, updated to reflect your Project Root Directory selection. Click the **Save** button.

Figure 23-11. Zona Modeler - Project Root Directory Selected



8 The main ZMUI screen redisplays.

9 Save your changes using the **File > Save** menu option.

Changing the Game and Audit Database Configuration

Zona Modeler stores the database parameters for a particular model file in the referenced configuration file. To change the database parameters, edit the referenced Zona Model configuration file using the ZMUI:

1 Select **File > Configure...**

2 The Configuration dialog displays:

3 Select either the **Game** or **Audit** tab.

4 The database parameters display in the **Database Configuration** sub-panel.

5 Change the old parameters to your new, desired parameters.

6 When you have finished making your changes, click the **Save** button.

7 Your new database parameters have been saved.

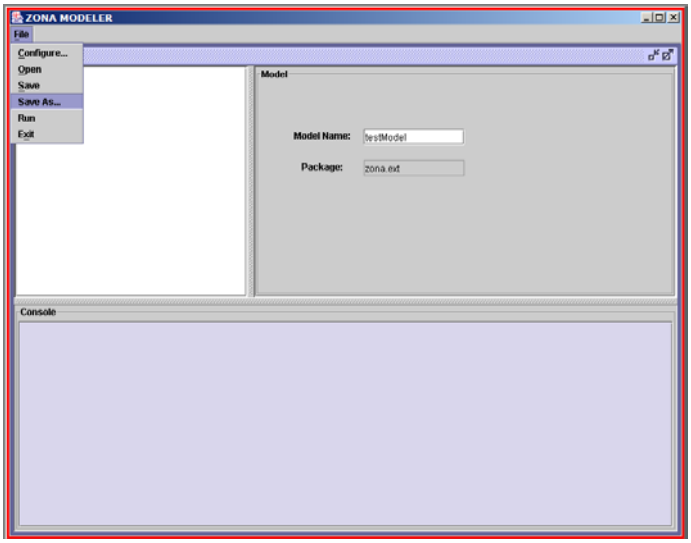
Within the Database Configuration sub-panel, the "Platform" drop-down displays a list of currently supported database platforms. Although Terazona can operate with any database platform that supports JDBC connectivity, only the platforms displayed in the drop-down have been tested and certified for optimum production performance and stability.

Renaming or Copying Your Model File

To rename (or to create another copy of) your model file from within ZMUI:

1 Select **File > Save As...**

Figure 23-12. Zona Modeler - Save As... Selected



2 The Select Zona Modeler XML File dialog displays.

3 Navigate to the directory where you want to save your new copy of the model file.

4 Enter the new model name into the **File Name** field.

5 Click the **Save** button.

6 Your model file (and associated configuration information) has been saved with your new specified name.

Creating Model Entities

ZMUI initially creates a blank model file with a single root node, **Model**. You can and should change this default root node name. The default root node is "empty" and contains no elements. The first thing you should do is add an Entity to the root node.

Within the ZMUI, you can add three types of Elements to any root node:

Table 23-5. ZMUI Elements

Entity Element	Description
Character	This corresponds to a Zona Server Character.
Child	This corresponds to a Zona Server Entity.
Guild	This corresponds to a Zona Server Guild.

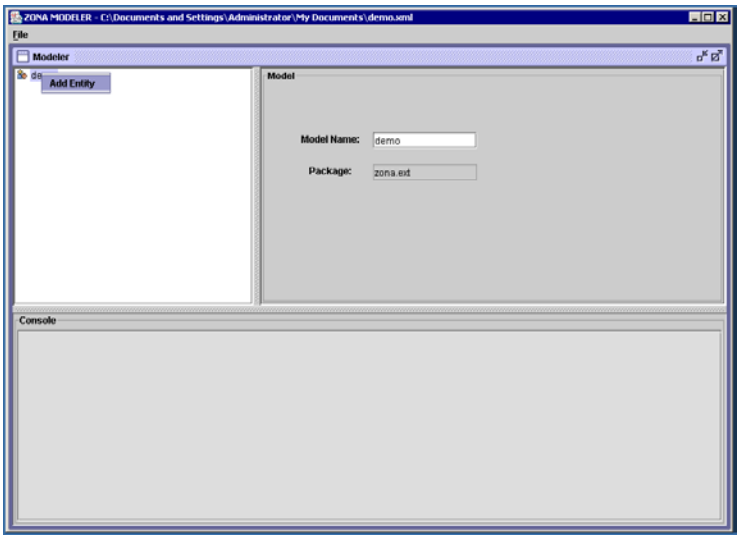
Adding a Character Entity

Character Entities are associated with Player Ids and extend **`ZonaServerCharacter`**. To create a Character Entity within ZMUI:

1 Use the mouse to highlight the root node element.

2 Option-click the mouse while hovering the cursor over the root node element. The **Add Entity** popup displays.

Figure 23-13. Zona Modeler - Add Entity



3 Either click or option-click to select the **Add Entity** action.

4 The **New Entity** dialog displays. The default Entity in the **Type** dropdown is "Character".

5 Click in the **Name** field to name your Character Entity.

6 Click the **OK** button to close the **New Entity** dialog.

7 The ZMUI display updates to display your newly created Character Entity.

8 Save your changes using the **File > Save** menu option.

Adding a Child Entity

Child Entities extend **`ZonaServerEntity`**. To create a Child Entity within ZMUI:

1 Use the mouse to highlight the root node element.

2 Option-click the mouse while hovering the cursor over the root node element. The **Add Entity** popup displays.

3 Either click or option-click to select the **Add Entity** action.

4 The **New Entity** dialog displays. Click the Type dropdown to display the available Entity choices.

5 Select the "Child" entry in the Entity **Type** dropdown list.

6 Click in the **Name** field to name your Child Entity.

7 Click the **OK** button to close the **New Entity** dialog.

8 The ZMUI display updates to display your newly created Child Entity.

9 Save your changes using the **File > Save** menu option.

Adding a Guild Entity

Guild Entities extend **`ZonaServerGuild`**. To create a Guild Entity within ZMUI:

1 Use the mouse to highlight the root node element.

2 Option-click the mouse while hovering the cursor over the root node element. The **Add Entity** popup displays.

3 Either click or option-click to select the **Add Entity** action.

4 The **New Entity** dialog displays. Click the Type dropdown to display the available Entity choices.

5 Select the "Guild" entry in the Entity **Type** dropdown list.

6 Click in the **Name** field to name your Guild Entity.

7 Click the **OK** button to close the **New Entity** dialog.

8 The ZMUI display updates to display your newly created Guild Entity.

9 Save your changes using the **File > Save** menu option.

Designing Model Entities

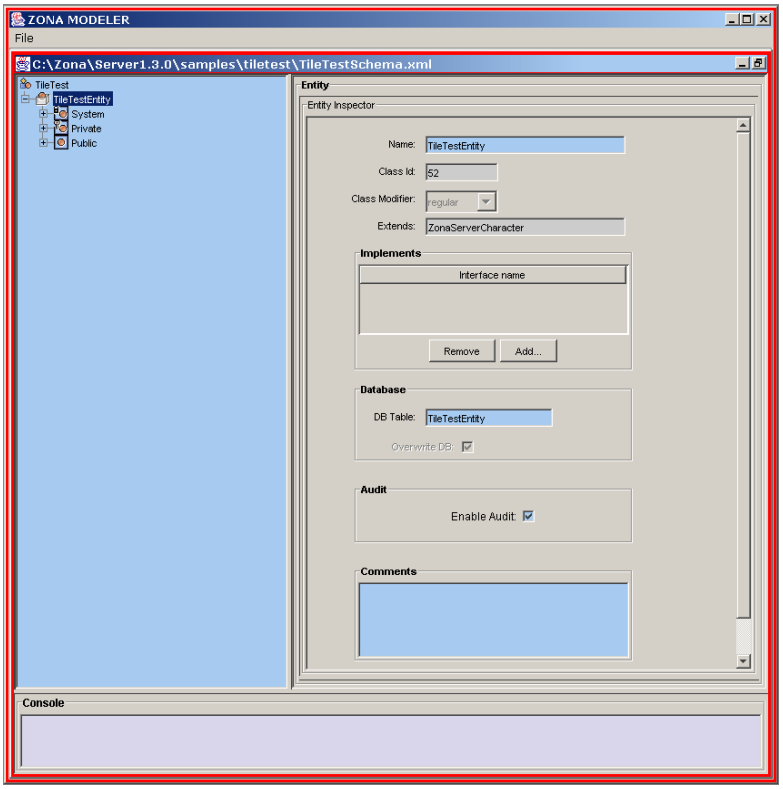
Zona Modeler creates initial Entities with three default Entity Properties:

Table 23-6. Default Entity Properties

Entity Property	Description
System	This corresponds to the Terazona System Property. Initially, there are no System Property Elements. You can add System Property Elements using the ZMUI.
Private	This corresponds to the Terazona Private Property. Initially, there are no Private Property Elements. You can add Private Property Elements using the ZMUI.
Public	This corresponds to the Terazona Public Property. Zona Modeler auto-generates an array of Public Property Elements. These contain auto-generated non-editable Public Property Element data, required for Terazona operation. You can add additional Public Property Elements using the ZMUI.

The default Entity Properties display within the ZMUI's Object Tree top-left panel:

Figure 23-14. Zona Modeler - Default Entity Attributes



Examining the Entity Attributes

The top-left Entity Inspector panel displays the Entity Attributes for each Entity selected in the top-right Object Tree panel, and also enables you to edit the Entity Attribute values. The un-ghosted Entity Attribute values are editable while some Attribute values are aare ghosted and non-editable (that is, these are auto-generated by Zona Modeler).

There are five Entity Attributes:

Table 23-7. Entity Attributes

Entity Property	Description
Entity Name	Defines the name of this Entity. Editable.
Class Id	The Class Id embedded by Zona Modeler within the run-time Entity object instantiation meta data. Non-editable.
Class Modifier	Modifies the auto-generated Zona Modeler output classes with one of the Java-dervived inheritance/subclassing control definitions: **regular** - standard class **abstract** - You cannot instantiate an object of this class, but you can subclass. **final** - You can instantiate an object of this class, but you cannot subclass.
Extends	Describes which category of Terazona API class this class extends from.
Implements	Defines which Interface this Property implements.
DB Table	Describes the name of the Table used to persist this class within the Zona Modeler database.
Audit	This defines whether the Auditing Server will audit any of this Property's Elements. This must be selected to enable the auditing of any of the Property's Elements.
Comments	Defines text comments for selected Entities or Properties. These comments do not affect the game logic.

Examining the Entity Property Attributes

All Zona Modeler Entity Properties exhibit six Entity Property Attributes:

Table 23-8. Entity Property Attributes

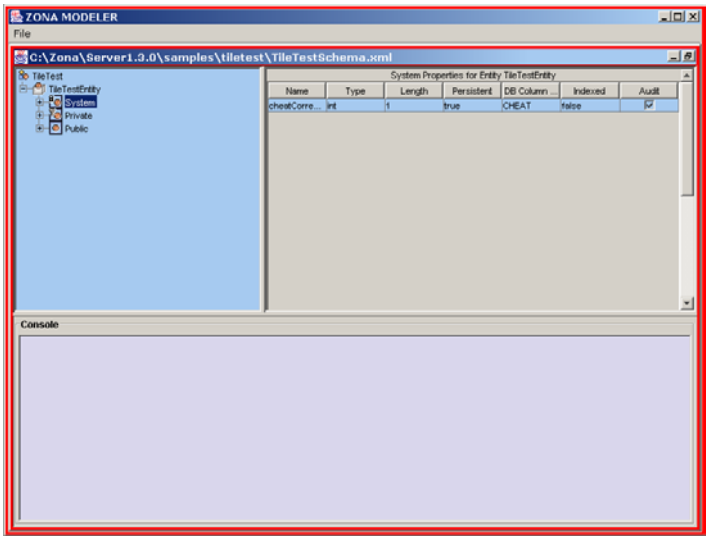
Entity Property Attribute	Description
Name	This identifies the Terazona API name for this class of Property Elements.
Type	This defines the type of this class of Property Elements.
Length	This defines the length of this class of Property Elements, in units of Type.
Persistent	This defines whether this class of Property Elements should be persisted to the Game Database.
DB Column Name	This defines the column identifier used within the Zona Modeler database to reference this class of Property Element.
Indexed	This defines whether Zona Modeler will use this class of Property Elements as a foreign key for queries.
Audit	This defines whether the Auditing Server will audit this Property. Auditing must have been selected within the parent Property to enable the auditing of any of the Property Elements.

The sub-properties of the Entity Properties, or the Entity Property Elements, also use the same Attributes. See *Displaying Entity Property Elements on page 149*.

Displaying Entity Property Attributes

To display the Entity Property Attributes for a specific Zona Modeler Entity, select the Entity Property in the Object Tree top-left panel. The top-right Inspector panel updates to display the Entity Property Attributes.

Figure 23-15. Zona Modeler - Public Property Attributes Displayed



Examining the Entity Property Elements

For each type of Entity created, Zona Modeler auto-generates an array of type-specific Entity Property Elements with non-editable data values. The Entity Property Elements correspond to Terazona API member function parameters, and enable you to examine and define the Entity Property Element Characteristics, which use the same six definitions as the Entity Property Characteristics described in *Entity Property Attributes on page 145*.

There are nine default Character Entity Public Property Elements:

Table 23-9. Character Entity Public Property Elements

Entity Property Elements	Description
entityId	The Entity Id parameter definition for the Character Entity Public Property.
parentId	The parentId parameter definition for the Character Entity Public Property.
entityType	The entityType parameter definition for the Character Entity Public Property.
userRole	The userRole parameter definition for the Character Entity Public Property.
regionId	The regionId parameter definition for the Character Entity Public Property.
isMaster	The isMaster parameter definition for the Character Entity Public Property.
userId	The userId parameter definition for the Character Entity Public Property.
name	The name parameter definition for the Character Entity Public Property.
entityInfo	The entityInfo parameter definition for the Character Entity Public Property.

There are seven default Child Entity Public Property Elements:

Table 23-10. Child Entity Property Elements

Entity Property Elements	Description
entityId	The Entity Id parameter definition for the Child Entity Public Property.
parentId	The parentId parameter definition for the Child Entity Public Property.
entityType	The entityType parameter definition for the Child Entity Public Property.
userRole	The userRole parameter definition for the Child Entity Public Property.
regionId	The regionId parameter definition for the Child Entity Public Property.
isMaster	The isMaster parameter definition for the Child Entity Public Property.
entityInfo	The entityInfo parameter definition for the Child Entity Public Property.

There are five default Guild Entity Property Elements:

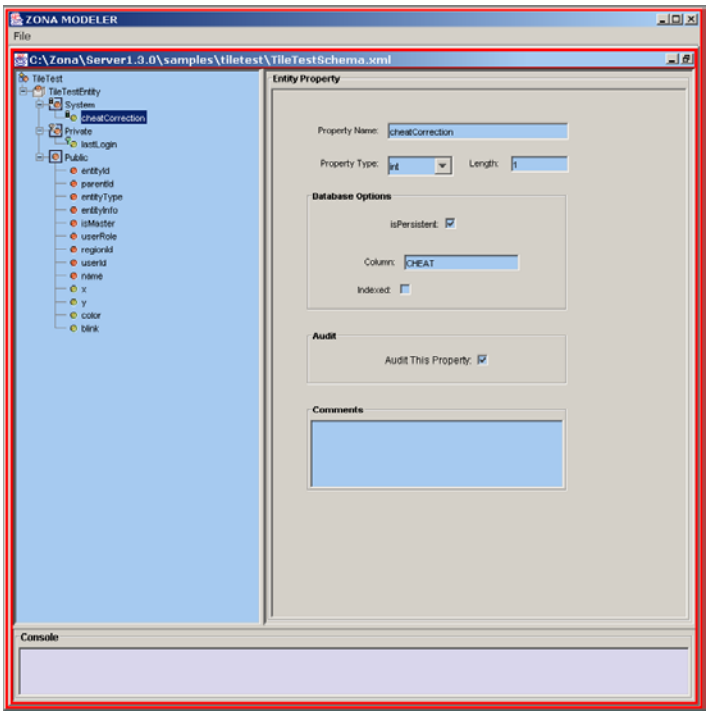
Table 23-11. Guild Entity Property Elements

Entity Property Elements	Description
guildId	The guildId parameter definition for the Guild Entity Public Property.
propertyAttributes	The guildId parameter definition for the Guild Entity Public Property.
inviterAttributes	The guildId parameter definition for the Guild Entity Public Property.
guildName	The guildId parameter definition for the Guild Entity Public Property.
entityInfo	The guildId parameter definition for the Guild Entity Public Property.

Displaying Entity Property Elements

To display the Property Elements for a specific Zona Modeler Entity, select an Entity Property in the Object Tree top-left panel. If it is not "open", click the associated ⊕ "twisty" icon to display the list of Property Elements. Select an Element by clicking on it. The top-right Inspector panel updates to display your selected Entity's Property Element Attributes:

Figure 23-16. Zona Modeler - Entity System Property Element Attributes

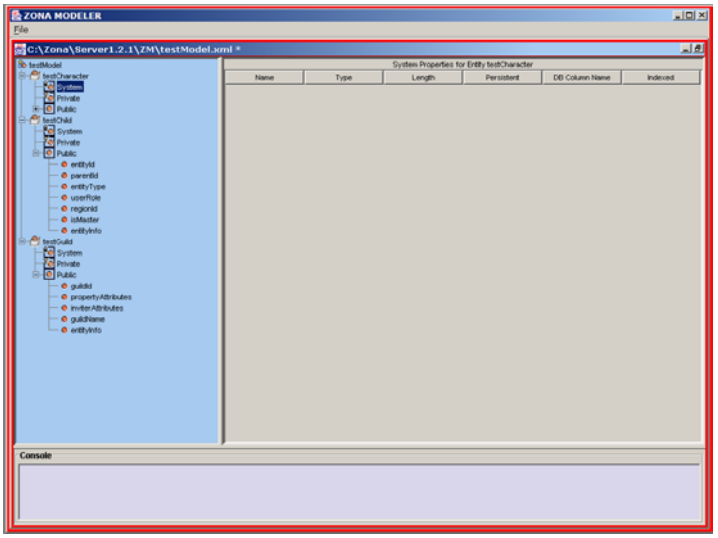


Adding Entity Property Elements

Zona Modeler enables you to non-programmatically create and modify Public, Private, and System Property Elements of Game Objects. Instead of tinkering with **`struct`** and **`class`** definitions, you can use Zona Modeler to refine and optimize your in-game data structures. To add an Entity Property Element:

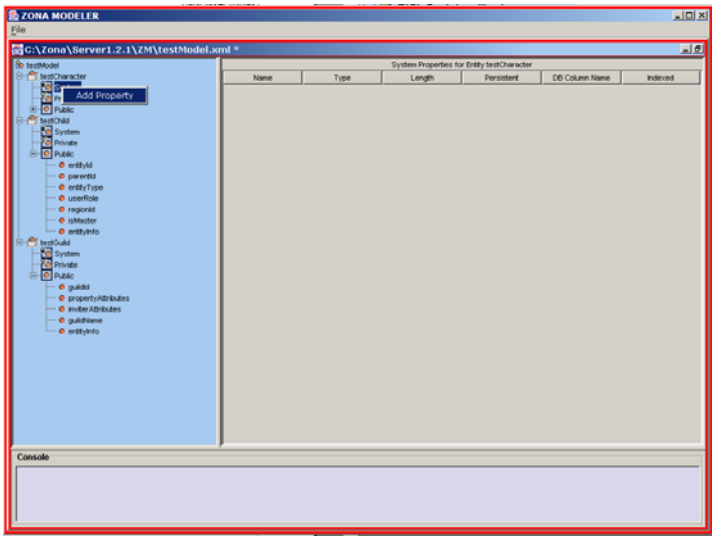
1 In the top-left Object Tree panel, select your desired Entity Property by highlighting it with a mouse click or the keyboard.

Figure 23-17. Zona Modeler - Selecting an Entity Property



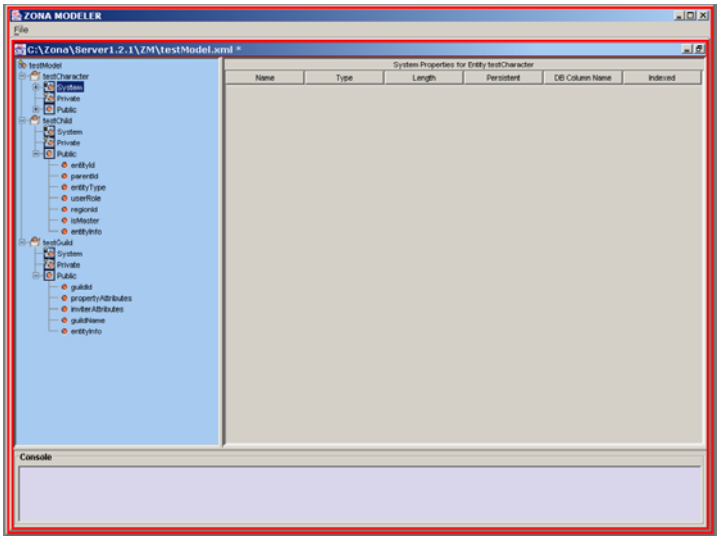
2 Option-click the selected Entity Property. The **Add Property** popup menu item displays.

Figure 23-18. Zona Modeler - Adding an Entity Property Element



3 Click the **Add Property** menu item. The Object Tree panel updates to reflect your Entity Property Element addition.

Figure 23-19. Zona Modeler - Entity Property Element Added



To display the newly added Entity Property Element, click the associated ⊕ "twisty" icon as described in *Displaying Entity Property Elements on page 149*.

Examining Entity Property Element Attributes

When created, Entity Property Element Attributes contain these default values:

Table 23-12. Default Entity Property Element Attributes

Entity Property Attribute	Description
Property Name	The name of the Entity Property Element. Default: **property**.
Property Type	The type of this class of Entity Property Element. The options are: **boolean** - *default* byte char short int long float double
Length	The length of this class of Entity Property Elements, in units of Property Type. Default: **1**
isPersistent	Defines whether this class of Property Elements should be persisted to the Game Database. Default: **YES**
Column	Defines the column identifier used within the Zona Modeler database to reference this class of Entity Property Element. Default: **property**
Indexed	Defines whether Zona Modeler will use this class of Property Elements as a foreign key for queries. Default: **NO**

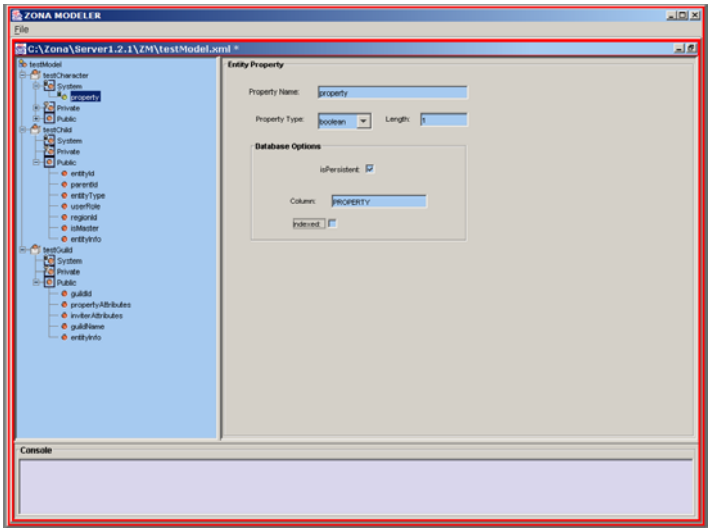
Modifying Entity Property Element Attributes

1 To modify the Property Element Attributes for a specific Zona Modeler Entity Element, select an Entity Property in the Object Tree top-left panel.

2 If it is not “open”, click the associated ⊕ “twisty” icon to display the list of Entity Property Elements. Select an specific Element by clicking on it.

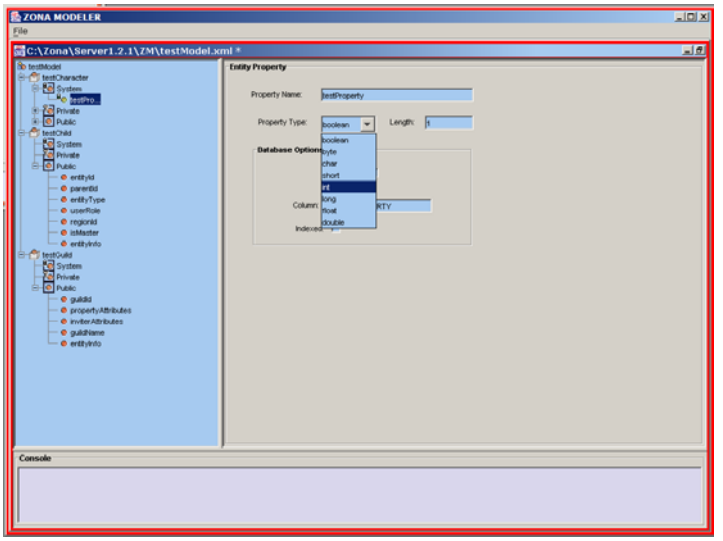
3 The top-right Entity Inspector panel updates to display your selected Entity’s Property Element Attributes:

Figure 23-20. Zona Modeler - Displaying an Entity’s Property Element Attributes



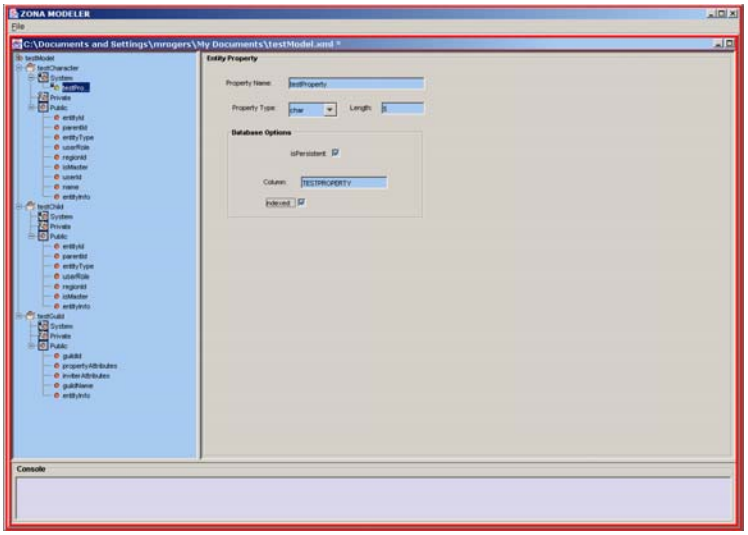
4 Select the field(s) you want to modify. Type a new value, select a new value from the drop-down menu, or check/uncheck the radio boxes.

Figure 23-21. Zona Modeler - Modifying an Entity's Property Element Attributes



5 When you have finished making your desired changes, click on the new Entity Property Element in the top-left Object Tree panel. The display updates to reflect your changes

Figure 23-22. Zona Modeler - Displaying a Modified Entity's Property Element Attributes



Compiling Model Entities

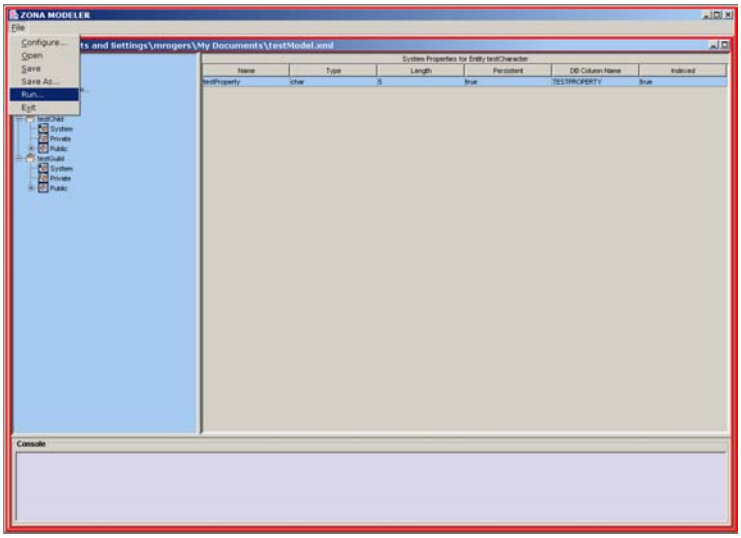
During the design-time phase of Zona Modeler UI, you create and modify Zona Model Entities to represent Network Game Objects. During the run-time phase of Zona Modeler UI, you command Zona Modeler to compile your Zona Model Entities and produce output code and metadata.

During this run-time compilation phase, Zona Modeler combines your Model File with its constituent Entities and database references to auto-generate C++ and Java code binaries that, combined with auto-generated database information, is suitable for Server-side run-time object instantiation and Client-side project linking and embedding.

Running Zona Modeler

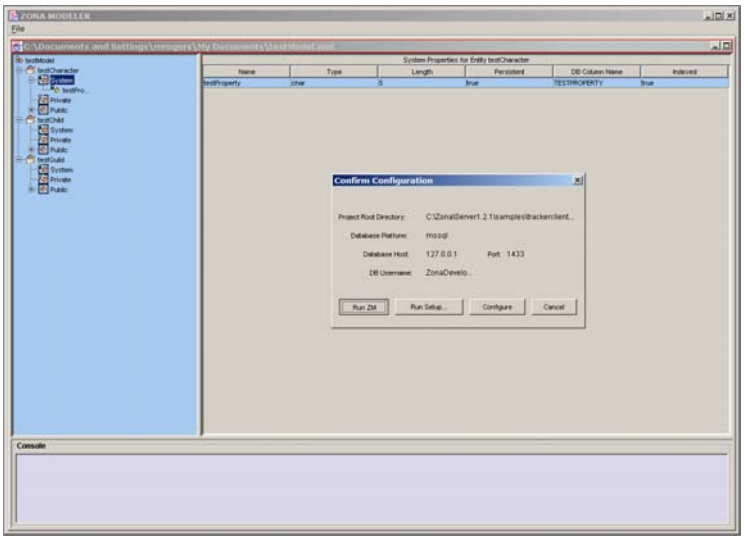
1 Select **File > Save As...**

Figure 23-23. Zona Modeler - Menu Item "Run" Selected



2 A Confirm Configuration dialog displays, giving you the option to modify the database configuration (**Configure**), run a batch file (**Run Setup...**) or to compile the Model Entities (**Run ZM**). Click the **Run ZM** button to continue.

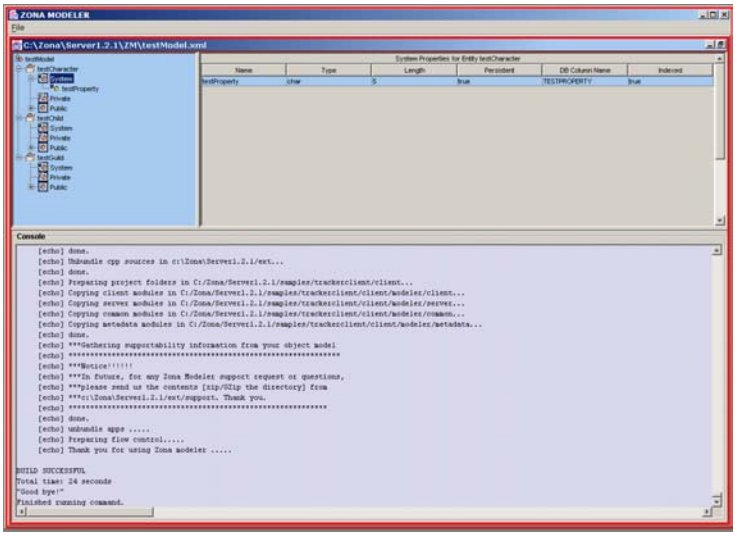
Figure 23-24. Zona Modeler - Confirm Configuration Dialog



3 Zona Modeler compiles your Model Entities and displays its progress within the Console window. You can scroll through the command output to check for errors,

inconsistencies, or to monitor Zona Modeler's outputs and output locations. A successful build completes with the message "BUILD SUCCESSFUL".

Figure 23-25. Zona Modeler - Successful Build Completed



Zona Modeler creates a specific output file and directory structure. You should maintain this generated directory structure at all times, even inside your project directories:

```
modeler
        client
        common
        metadata
        server
```

Chapter

24

Character Entity Object

This chapter explains the Character Entity Object (CharEO), the critical "unit" of Terazona in-memory game object representation. CharEOs also enable persistence and rollback, because the GSSs read and write CharEOs as binary data to and from the Game Database and as normalized data to and from the Audit Database.

Examining the Character Entity Object

The Character Entity Object (CharEO) is the fundamental transfer unit between the GSSs and the Game Database. Every CharEO is associated with a unique, persistent Entity Id. This can be used to relate the binary data stored within the Game Database to the in-memory Entity and Character structures manipulated by the GSSs and communicated among GSSs and between GSSs and Clients.

Every CharEO has a single Master GSS that owns the primary copy of that data. Any GSS can use the **`ZonaServerCharacter::getMaster()`** function to check for ownership.

The CharEO also contains Entity-specific and header data such as timestamps, Class Ids, Entity Type information, and various system flags and memory allocation data used by the GSSs for transaction processing and bandwidth optimization.

Understanding the Character Entity Properties

The CharEO is the complete binary representation of an Entity or Character. The Character Properties are a data structure subset of the CharEO comprising the Public Properties and the Private Properties that enables you to selectively modify and publish data between Clients and GSSs. There are three Character Entity Properties:

Table 24-1. Character Entity Properties

Property	Description
Public Properties	Repository for public data such as display textures, location, or size that are published to all GSSs and Clients.
Private Properties	Repository for private Client-specific data such as ammunition that are published to all GSSs but only to the owning Client.
System Properties	Repository for private data such as curses or secrets that are published to all GSSs but never to any Clients (including the owning Client.)

Only the GSSs have complete access to all the data within a CharEO, including the System Properties. As a GSS Plugin developer, you control the validation of information requests from Clients and therefore control Clients' access to the information contained within their CharEOs and other Clients' CharEOs.

The NPC Server is a Trusted Client and receives Entity and Character Updates the same way as other, non-trusted Clients. However, the bandwidth available between NPC Servers and GSSs is typically several hundred times greater than the bandwidth available between Clients and GSSs. Your NPC Server can perform data-instensive operations that would not function well for other, non-LAN Clients.

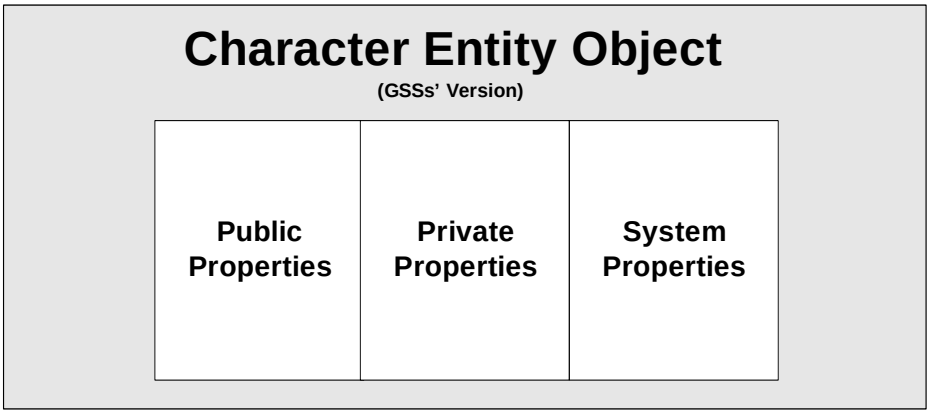
The default behavior of Terazona's GSSs is to publish all the Public Properties to all Clients, to publish the Private Properties only to the owning Client (and never to non-owning Clients), and finally to deny absolutely any Client access to their (or other Clients') System Properties.

The Master GSS Plugin can directly modify the Character Properties (CharProps) of CharEOs that it owns, and then publish these changes to other GSSs that store Ghost copies of that CharEO. To modify the Master copy of a CharEO owned by another GSS, the GSS Plugin sends a modification request to the owning GSS, which then validates this request. When GSSs modify their CharEOs, the GSS Plugin developer can choose to save these changes to the Game Database or to propagate the changes to any or all of the managed Clients.

The truncation of the CharEO throughout the Terazona system safeguards Client security and privacy, protects system integrity, and reduces bandwidth requirements. The following illustrations demonstrate how each system in Terazona receives a different version of the CharEO.

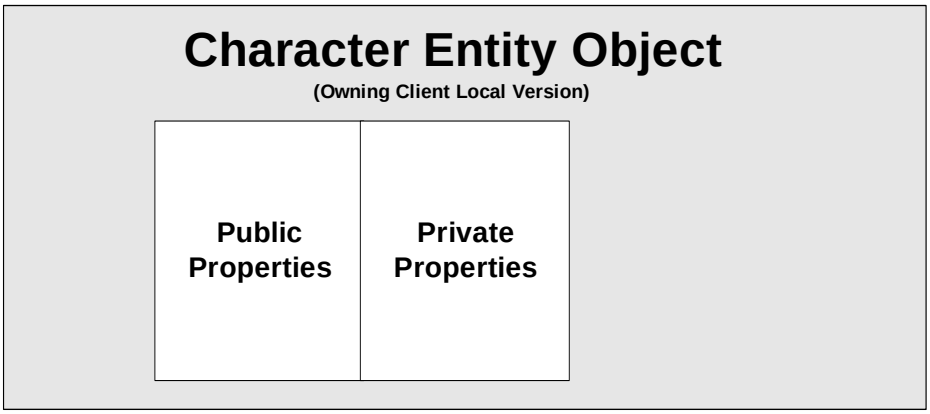
All GSSs maintain the most complete version of the CharEO. The managing GSS maintains the Master CharEO while other GSSs that have subscribed to entity updates from that Client (because of Regional proximity or in-game activities) maintain a Ghost CharEO. The managing GSS publishes a subset of the CharEO to its managed Client that omits the System Properties. The other GSSs publish a subset of their Ghosted CharEOs to their managed Clients that contains only the Public Properties.

Figure 24-1. Character Entity Object - GSS's Version



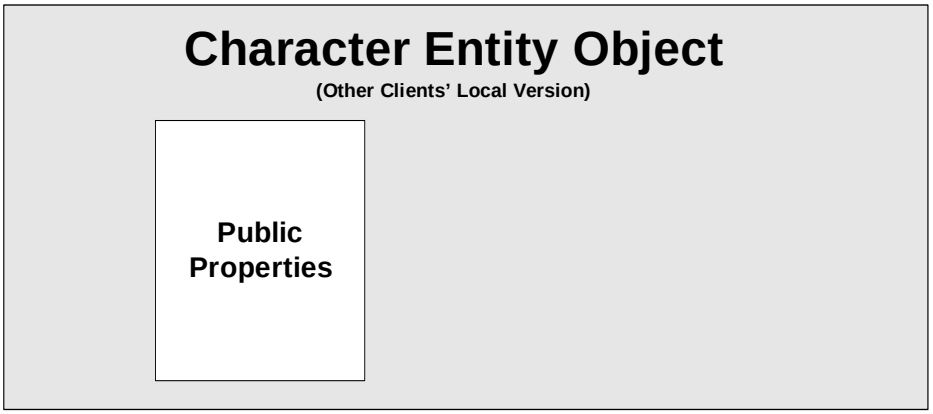
The owning Client does not receive a local copy of its System Properties. From a Player's point of view, they may suspect that their Character possesses some attributes or "curse" that is affecting their interactions with other Characters, but they have to deduce this from second-hand information.

Figure 24-2. Character Entity Object - Owning Client Local Version



The non-owning Clients receive only that portion of another Client's CharEO necessary to display that Client within their game environment. This lowers Client<->Server bandwidth requirements and protects against Client-side hacking and packet snoop attacks.

Figure 24-3. Character Entity Object - Other Clients' Local Version



Managing the Character Properties

When you instantiate either a **`ZonaServerCharacter`**, **`ZonaClientCharacter`**, or **`ZonaClientEntity`** object, you gain access to an array of auto-generated object-specific getter and setter methods, along with appropriate utility functions. During its code generation stage, Zona Modeler auto-generates these getter and setter methods for Entity properties defined within the Zona Model file.

Following Character or Entity instantiation, you use the **`get`*`XXXX`*`()`** functions to access the property data (where ***`XXXX`*** is a Public, Private, or System Property developer-defined within the Zona Model file). Then you use the **`set`*`XXXX`*`()`** functions to modify the property data.

The Zona Entity Manager (ZEM) maintains synchronization between objects on the GSS and attached Clients. For both Client-side and Server-side Property modifications, the ZEM maintains a "dirty list" of data that require updating or propagation. You do not need to explicitly tag Properties or data structures for distribution. At the end of a game loop, you call the **`ZonaClientCharacter::publish()`** function to trigger ZEM to bring the object data up to date. The distribution process is transparent: the Zona Entity Manager (ZEM) handles this activity in the background, optimizing bandwidth and message transfer between game objects.

You can check whether a Client has permission to change specific properties using the **`ZonaEntityValidate::onValidateEntityPropertyUpdate()`** function.

Managing the Public Properties

The Public Properties contain all the data relevant to:

- The display of an Entity or Character within the game environment.
- The state of an Entity or Character within the game environment.

This Data is sent to all subscribed Clients and GSSs. Good candidates for inclusion within the Public Properties are:

- Character or Vehicle type – usually stored as an object Id
- Children of character or vehicle – that is, weapons, books, scrolls, ammunition that you want to store as lists and not go through the overhead of Entity instantiation.
- Character Name or physical attributes.
- Health State – visual level of damage, and so on
- Clothing - usually as an object Id
- Emotional State
- Position

- Orientation
- Velocity
- Level or Prestige Ranking

The Public Properties are used for updating Character data that changes rapidly and must be propagated quickly to other Clients. These elements are generally updated by the Client and sent to the GSS for validation and redistribution within the Terazona cluster. This offloads the CPU burden from the GSSs to the Clients, distributing the processing among the Players' machines.

Managing the Private Properties

The Private Properties contain all the data relevant to the attributes of an Entity or Character within the game. This data is sent to the managing Client and all subscribed GSSs. Good candidates for inclusion within the Public Properties are data that a Player should know about their Character but that other Players should not know, such as:

- Ammo
- Level
- Dexterity
- Constitution
- Wisdom
- Hit Points
- Skills

The Private Properties are used for updating Character data that changes occasionally and must only be shared between a Client, its managing GSS, and other GSSs with no onward distribution to other Clients (except for game-specific GSS-validated functions such as a "Reveal" spell or "Espionage" activity). These elements are generally updated by the GSS and sent to the Client and other GSSs.

Managing the System Properties

The System Properties contain all the data relevant to the attributes of an Entity or Character within the game that should be kept secret and not disclosed to any Clients (except for game-specific GSS-validated functions such as a "Reveal" spell or "Espionage" activity). This data is never sent to the managing Client but is distributed to all subscribed GSSs. Good candidates for inclusion within the System Properties are data that a Player should not know about their Character but that subtly affects their other Properties such as:

- Curses
- Geas
- Spells
- Armor Damage Modifiers
- Age

The System Properties are used for modifying Character data and storing "system secrets" and is only shared between GSSs with no onward distribution to any Clients. These elements are generally updated by the Managing GSS and sent to the other GSSs.

Updating the Character Properties

1. Upon the Creation of Client A
   a. The Character Entity Object is created and initialized.
   b. The GSS validates this Entity Object's creation.
   c. The Public Properties are published to Client A.
   d. The Private Properties are published to Client A.
2. When Client A Enters Client B's Sphere:
   a. Public Properties of Client A are published to Client B
   b. Public Properties of Client B are published to Client A.
3. When Client A Moves in Client B's Sphere of Interest (SOI):
   a. Client A's Public Properties are published to Client B.
   b. Client B's Public Properties are published to Client A.
4. When Client A Leaves Client B's SOI:
   a. Client B is informed that Client A has exited B's SOI.
   b. Zona Entity Manager on Client B's Managing GSS purges Ghost copies of Client A's CharEO.
   c. Client A is informed that Client B is no longer within its SOI.
   d. Zona Entity Manager on Client A's Managing GSS purges Ghost copies of Client B's CharEO.

Chapter

25

Simple Client Creation

This chapter demonstrates some of the CAPI functionality using the TrackerClient sample application.

Creating a Simple Terazona C++ Client

The TrackerClient sample application demonstrates how to create a simple Terazona Client. After you review this sample, you can use extend this simple Client to begin creating more complex Terazona Clients.

Tracker Client

This code example demonstrates several key components of the client side network process for an online game:

- Initialization of ZonaServices
- Player Login
- Selecting a Character
- Registering a Callback for Game State Monitoring
- Entering a Character into the Game
- Subscribing to the Game State
- Sending Game State data to the server
- Exiting the Character from the Game
- Player Log Off

Reviewing the code

As with any program, you need to write the code. However, creating a program to pass messages is almost as easy as writing the classic "Hello World" program. This walkthrough will take you through the basic steps using code fragments from the **`TrackerClient.exe`** program installed with the client development option. You can find the source code to the **`TrackerClient.exe`** in this directory:

```
%ZONA_HOME%\samples\TrackerClient\client\
```

The command-line invocation for TrackerClient will be:

```
TrackerClient username password http://host:port
                              messageFrequency NoOfPackets
```

Instantiating ZonaServices

All Terazona Clients initialize and instantiate a ZonaServices object that provides Clients with functions to login, logout, chat, set up callbacks to receive game state updates from other clients and server-side Entities.

The ZonaServices object requires no argument for creation. Example:

```
zonaServices* zonaServices;
zonaServices = new ZonaServices(); // == new ZonaServices
                                   // (true, false)
```

Logging In

Creating the Terazona services object does nothing by itself it in terms of connecting clients to servers. To acquire a session on the server, use the **`ZonaServices::login()`** function:

```
int ret_code = zonaServices->login( argv[3],
    argv[1], argv[2]);
    if ( ( ret_code != ERR_SUCCESS) ) {
      printf("Error logging onto Terazona server.
      exiting.\n");
    exit(ret_code);
}
```

The login function authenticates against the Game Database.

Understanding the GSS Event Sequence During Login

Client calls the Login() function

The Client sends off the username/password to the Dispatcher and it receives back the least-loaded GSS (actually the message broker, or MB) URL to enable front-end load balancing. This is transparent to the Client.

Dispatcher Reroutes Player to Least-Loaded GSS

The Dispatcher determines which GSS to use by checking the number of players on each GSS as well as processor capability (the load can also depend on the number of messages being managed by the GSS at that moment). The Client then disconnects from the dispatcher's MB and reconnects to the GSS MB.

Managing the Character

You must create Client-side **ZonaClientCharacter** objects. These are linked with the **ZonaServerCharacter** objects maintained by the GSSs. The **TrackerCharacterClient** class is auto-generated by Zona Modeler from the definitions within the **TrackerSchema.xml** model file.

```
ZonaClientCharacterPtrVector characters;
TrackerCharacterClient *character;
```

Getting the Characters

TrackerClient attempts to pull Character data from the Game Database. If there is no Character data there that corresponds to the login credentials (that is, a zero **characters** array) it creates an initial, default Character):

```
zonaServices->getCharacters(characters);
if ( characters.getSize() <= 0 ) {
    char* theLoginName = argv[1];
    int theCharacterNameLength = 2*strlen(theLoginName) + 1;
    char* theCharacterName = new
                              char[theCharacterNameLength];
    TrackerCharacterClient* zcc = new
                                  TrackerCharacterClient();
    sprintf(theCharacterName, "%s%s",
                               theLoginName, theLoginName);
    zcc->setName(theCharacterName, theCharacterNameLength);
    zonaServices->createCharacter(*zcc);
    character = zcc;
    delete theCharacterName;
}else{
    //Select first character
    character = (TrackerCharacterClient*)characters[0];
}
```

During the creation, the code populates the Character Name using the **set*Name*()** function that has been auto-generated by Zona Modeler.

Selecting the Character

Selecting a Character means using the **ZonaClientCharacter::select()** function to notify the GSS which is the active Character. The GSS will populate its Server-side ZonaServerCharacter object cache appropriately. The code to do this is very simple:

```
// select a character
character->select();
```

Entering the Character

Now that you have created an Active Character through Selection, it is time to place the Character within the game world.

You use the **`ZonaClientCharacter::enter()`** function to place the active Character within the game. This alerts the GSS to start tracking the Character.

```
character->enter();
```

Before you enter a Character you must first Register for Game State Updates. See *Monitoring a GameState Callback on page 175* for details.

Managing the Game State

After you create a Client Character and **before** you enter it within the game, you must implement a GameState Callback class that registers with Terazona to receive the GSS dynamic data updates.

The most basic callbacks to implement are for when a Character enters or leaves a Sphere of Interest, when it receives a Game State Message, or when a Client has been forcibly disconnect from a GSS.

Creating a GameState Callback

Implement a Callback object to receive game state messages:

```
class TrackerClient : public GameStateCallback,
                                        public EntityCallback
{
public:
    // this Game State callback is designed to
                                    receive gamestate messages
    // Messages are sent in the main() function below
                        from the ZonaServices::SendGameState()
  void onReceivedGameStateMsg
             (int entityId, char* stateData, short dataSize)
    {
        //Casts byte buffer into the a data structure
        MSG_UPDATE* state = (MSG_UPDATE*)stateData;
        //Log of message recieved by client
                                implemented gamestate receives
      printf("Received senderId=%i entityId=%i X=%f Y=%f\n",
       entityId, state->entityId, state->posX, state->posY);
    }
   void onNotifyEntityPropertyUpdate(ZonaClientEntity* zce)
{
        TrackerCharacterClient* myEntity;
        myEntity = (TrackerCharacterClient*)zce;
        printf("Received:  property update: x: %d y: %d\n",
               myEntity->getPosX(),
               myEntity->getPosY());
    }
    // This callback is fired when an entity ENTERS
                                        the sphere of the user
    void onNotifyEntityJoinedSphere(ZonaClientEntity* zce)
    {
        printf("\nentityJoinedSphere:
```

```
                playerEntityId=%d\n", zce->getEntityId());
    }
    // This callback is fired when an entity LEAVES
                                        the sphere of the user
    void onNotifyEntityDepartedSphere(ZonaClientEntity* zce)
    {
        printf("\nentityDepartedSphere:
                  playerEntityId=%d\n",zce->getEntityId());
    }
    void onPlayerReset(int action , int error){
        printf("User has been kicked off my server
                      action=%d error=%d\n", action, error);
    }
};
```

Monitoring a GameState Callback

Having created a suitable GameState Callback with logic to respond to your selected game data updates, you must activate the GameState callback monitoring function within the main body of TrackerClient.

Within the **`main()`** function, you attach a Game State callback (for event data updates from the GSS) and an Entity Update callback (for Property updates from the GSS) within the ZonaServices object:

```
    // register the game state callback object
    // (defined above) to receive the messages
    TrackerClient* callback = new TrackerClient();
    zonaServices->monitorGameState(callback);
    zonaServices->monitorEntityUpdates(callback);
```

The **`monitorGameState()`** function is used primarily as a Character event update channel, and the **`monitorEntityUpdates()`** function as a Character data update channel.

Subscribing to GameState Updates

After the Client has registered the GameState Callback, you can subscribe the Client to all Terazona gamestate messages (or a subset of a particular class of gamestate messages) about that Client using the **`ZonaServices::subscribeToGameStateMsgs()`** function:

```
zonaServices->subscribeToGameStateMsgs
                    (character->getEntityId(),ZONA_MSG_ALL);
```

Message Filters for subscriptions can be updated dynamically during game execution to optimize Server<->Client bandwidth usage.

Communicating with the Server

With the callbacks in place you can send and receive Property updates and Game State Messages.

To broadcast the active Character's "dirty" Property data to the GSS, use the **`ZonaClientCharacter::publish()`** function.

To send game state message updates, use the **`ZonaServices::sendGameStateMsg()`** function. To receive game state messages, use the **`ZonaServices::processMessage()`** function.

This code example simulates the transmission of 2-dimensional positional data.

```
for (float i=0.0; i<10.0; i++)
    {
    for (float j=0.0; j<numMsg; j++)
      {
      // set the 2d positional data
      state.posX = i;
      state.posY = j;
      // send the data to the Terazona Game State Server
      printf("sending X=%f Y=%f\n", state.posX, state.posY);
      if ( propertyUpdate ) {
          character->setPosX(state.posX);
          character->setPosY(state.posY);
          character->publish();
      }
      else
      // Main call that sends game state to server
      zonaServices->sendGameStateMsg((char *) &state, size);
      // process the messages that were sent by the server.
      while(zonaServices->getMessageBufferCount() > 0
          zonaServices->processMessage();
      // wait for half second so demo doesn't end at once
      Sleep(msgFreq);
    }
}
```

The Property modification functions **`TrackerCharacterClient::setPosX()`** and **`TrackerCharacterClient::setPosY()`** are auto-generated by Zona Modeler.

Leaving a Game

Leaving a game is a two-stage process for Characters. First they must exit the game world. Then they must end their terminate their ZonaServices session. This finalizes their exit and ensures correct CHaracter cleanup. The sequence is:

1. **ZonaClientCharacter::exit()**
2. **ZonaServices::logoff()**

Exiting the Game World

```
// remove the player from the sphere
character->exit();
```

Logging Off

```
// remove the player from the game
zonaServices->logoff();
```

For details on how to set up a compilation and development environment for Terazona projects, please see *Development Environment on page 109*.

Chapter

26

Managing Players Using The Server

Client-side development is only one part of developing a Terazona application. Clients are managed on the Server-side, by one or more Game State Servers (GSSs). Developers program responses to Client-side calls and embed them within the GSSs using the GSAPI functions. To completely understand how the system handles this Character management, you must understand the Client-Server interaction.

This chapter describes interactions from the server-side viewpoint.

Managing Server-Side Characters

Characters are managed on the server-side by the Zona Entity Manager (ZEM), a transparent Entity caching subsystem that keeps track of active Entities and Characters and manages their memory and interactions. When you create and destroy Server-side Entities and Characters (sometimes in response to Client-initiated function calls), the ZEM handles the initialization and cleanup.

Entities are added to the ZEM immediately before the Client executes the **`EntityCallback::onNotifyEntityJoinedSphere()`** callback, and are deleted from the ZEM immediately after returning from a **`EntityCallback::onNotifyEntityDepartedSphere()`** callback. Do not attempt to explicitly delete Entities while they are being managed by the ZEM.

Entering the Game

1. Player logs on to the game.
   (Authentication request is sent to the Authentication Server)
2. GSS Creates Player session object.
3. GSS sends possible character data to Client (from Game Database).
4. Player selects/builds Character for game play.
5. GSS validates selection, populates Player's sessions object as Active Character.
6. Character enters the game.
7. GSS calls GSS **`ZonaEntityValidate::onEntityJoinedGame()`** function in GSS Plugin.
8. GSS localizes Character in game world with specific Region.
9. GSS triggers **`ZonaRegionValidate::onPlaceEntityInRegion()`** function in GSS Plugin.
10. GSS Plugin publishes Character location back to Client.
11. GSS triggers **`ZonaEntityValidate::onEnterEntity()`** function in GSS Plugin: the Entity is now "live" and being tracked.
12. In response to incoming Client property updates, the GSS will fire the **`ZonaEntityValidate::onValidateEntityPropertyUpdate()`** function within the GSS Plugin.
13. In response to incoming Client game state messages, the GSS will fire the **`ZonaGameStateValidate::onValidateGameStateMsg()`** function within the GSS Plugin.

This table illustrates the interaction between Client, GSS Plugin, and GSS during Character Entrance:

Table 26-1. Sequence For Character Entry

Client	GSS Plugin	GSS
`ZonaServices::login()`		
		Server creates a session object for the player
`ZonaServices::getCharacters()` get list of available characters		
		GSS retrieves character from database and passes Entity Object to the client
`ZonaClientCharacter::select()` from list		
		GSS populates session object with the selected character
`ZonaClientCharacter::enter()`		
		Calls `ZonaEntityValidate::onEntityJoinedGame()` in GSS Plugin
	`ZonaEntityValidate::onEntityJoinedGame()` executes.	
		Calls `ZonaRegionValidate::onPlaceEntityInRegion()` in GSS Plugin
	Implementation for `ZonaRegionValidate::onPlaceEntityInRegion()` should publish the starting location back to the entering character	
		Calls `ZonaEntityValidate::onEnterEntity()` in GSS Plugin
	`ZonaEntityValidate::onEnterEntity()` executes. Developer implementation should create a local entity object and publish notification back to Client.	

Authenticating the Player

The player is authenticated in the login stage before Character selection. This is done by the Authentication Server following a request by the Dispatcher. Following successful Authentication, a player session is then associated with the least-loaded GSS.

Placing the Character

This is the sequence for Character Placement.

Table 26-2. Character Placement Sequence

Client	GSS Plugin	GSS
`ZonaClientCharacter::enter()`		
		Calls **`ZonaEntityValidate::onEntityJoinedGame()`** in GSS Plugin
	`ZonaEntityValidate::onEntityJoinedGame()` executes.	
		Calls **`ZonaRegionValidate::onPlaceEntityInRegion()`** in GSS Plugin
	Implementation for **`ZonaRegionValidate::onPlaceEntityInRegion()`** should publish the starting location back to the entering character	
		Calls **`ZonaEntityValidate::onEnterEntity()`** in GSS Plugin
	`ZonaEntityValidate::onEnterEntity()` executes. Developer implementation should create a local entity object and publish notification back to Client.	
`EntityCallback::onEntityJoinedSphere()` is called.		

Table 26-2. Character Placement Sequence

Client	GSS Plugin	GSS
Entering client calls **`ZonaServices::sendGameStateMsg()`** to update data on server.		Detects Client game state messages, causes **`ZonaGameStateValidate::onValidateGameStateMsg()`** to fire in GSS Plugin
Other clients in Sphere start receiving game state via **`GameStateCallback::onReceivedGameStateMsg()`**	GSS Plugin can start calling publish functions to send data to other Clients and GSSs	

Exiting the Game (Logout)

This is the sequence for Character Exit & Player Logout:

Table 26-3. Sequence For Character Exit & Player Logout

Client	GSAPI	GSS
Call **`ZonaClientCharacter::exit()`**		
		Calls **`ZonaEntityValidate::onEntityDepartedGame()`** in GSAPI
	`ZonaEntityValidate::onEntityDepartedGame()` executes. Save lasts state to database.	
		Calls **`ZonaEntityValidate::onExitEntity()`** on Plugin to signal GSS is stopping tracking Character
	`ZonaEntityValidate::onExitEntity()` signals GSS to purge entity cache.	
Other clients in Sphere receive **`EntityCallback::onNotifyEntityDepartedSphere()`** and they can remove exited Character from their game environment.		
Call **`ZonaServices::logoff()`** to completely disengage Client		

Chapter

27

Managing Characters Using The Server

Character Management can be considered a separate issue from Player and Entity management because of specialized nature of dealing with Characters. Character Management entails creation, deletion, selection, storage and modification of players' game Characters. This chapter discusses in detail how the server handles the validation of the Character Entity.

Creating a Character

See below for the various steps taken by the client, the GSS Plugin and the GSS during Character creation.

Table 27-1. Sequence For Character Creation

Client	GSAPI	GSS
`ZonaServices::createCharacter()` defines a ZonaClientCharacter ZonaClientCharacter contains Name, EntityId, RegionId, ParentId, Character Properties.		
		Passes Character Properties to GSS Plugin via **`ZonaCharacterValidate::onValidate CreateCharacter()`**

	`ZonaCharacterValidate::onValidate CreateCharacter()`
	{
	Validate Char Data
	Return True/False
	}	

		Creates a new entry in the database Write Changed Property data to database if true Returns true or false.
Client checks for success or failure.		

Selecting a Character

To display a list of Characters selectable by a Client session, use this function:

```
zonaServices->getCharacters(characters);
```

The call to **`ZonaServices::getCharacters()`** returns the root Entities (that is, Characters) associated with the user's Login Id. It returns an empty **`ZonaClientCharacterPtrVector`** if there are no root characters. The children of the root Entities are not retrieved.

To select a specific Character in the **`ZonaClientCharacterPtrVector`** array, use the **`ZonaClientCharacter::select()`** function:

```
character = (TrackerCharacterClient*)characters[0];
character->select();
//Define unique entity ID
state.entityId = character->getEntityId();
```

It is the caller's responsibility to delete the objects pointed to by the pointers in the returned **`ZonaClientCharacterPtrVector`**. Don't delete the Character object that is to be selected!

Entities are added to the ZEM immediately before the Client executes the **`EntityCallback::onNotifyEntityJoinedSphere()`** callback, and are deleted from the ZEM immediately after returning from a **`EntityCallback::onNotifyEntityDepartedSphere()`** callback. Do not attempt to explicitly delete Entities while they are being managed by the ZEM.

Modifying a Character

Character data can be modified by both the client and the GSS Plugin. Which data gets modified by which component depends on the game design and the data itself. However, an example case is easily described here.

Data such as positional and orientation data is commonly modified by the user. Data such as health and wisdom are commonly modified by the GSS Plugin. Because the client has access to all of the these data attributes, it is the responsibility of the GSAPI to validate the character attributes during the course of the game. The separation of the Character attributes are described further in *Character Entity Object on page 159*.

Storing a Character

Once the Character is selected, it can be modified and updated. The **`ZonaClientCharacter::update()`** function updates the Entity Object for the character in the database and also the game server cache. The data is saved in the database as a data blob. This allows you to easily update and expand a Character Record without having to predefine the database schema.

Table 27-2. Sequence For Character Storage

Client	GSAPI	GSS
`ZonaClientCharacter::update()`		
		Processes Char Entity Object Passes Character Properties to GSAPI via **`ZonaCharacterValidate::onValidateUpdateCharacter()`** function

	`ZonaCharacterValidate::onValidateUpdateCharacter()`
	{
	Validate Entity Object
	Return True/False
	}	

		Updates the existing CharEO to database if true true or false. Sets new EntityId;
Client goes home happy if no errcode		

Deleting a Character

The character is deleted by simply calling the **`ZonaServices::deleteCharacter()`** function with the parameter being the character ID. This removes the character from the database.

Table 27-3. Sequence For Character Deletion

Client	GSAPI	GSS
`ZonaServices::` **`deleteCharacter`** **`(int characterId)`**		

	`ZonaCharacterValidate::`
	`onValidateDeleteCharacter()`
	{
	Validates request
	Return True/False
	}	

		Deletes the character in the database; returns errcode
Client goes home happy if no errcode		

Chapter

28

Simple Client-Server Demo Creation

This chapter illustrates how to create a simple client/server game called TileTest using Terazona.

Using TileTest

TileTest is a more advanced example of the client/server process. It includes both the client-side CAPI application and the server-side GSAPI DLL. This example will show how a developer can create a simple server-side GSAPI implementation as well as the client-side application.

TileTest Components

This code example demonstrates the same key components of the client-side network process in TrackerClient with the addition of a few others shown in italics

- Instantiation of ZonaServices
- Login
- *Creating a Character*
- Selecting a Character
- *Modifying the Character*
- *Entity Management with a simple Entity Manager*
- Entering a Character into the Game
- Registering a Callback for Game State Monitoring
- Subscribing to the Game State
- Sending Game State Data to the server
- Exiting the Game
- Logging off

In addition this system also demonstrates some server-side management:

- Game State Updates
- Simple Region Management
- Game State Validation
- Simple Entity Management

Reviewing the Code

You can find the source code to the **`TileTest.exe`** Client here:

```
%ZONA_HOME%\samples\tiletest\client
```

The server side code resides here:

```
%ZONA_HOME%\samples\tiletest\ServerPlugIn
```

The **`TileTestEntity`** class is auto-generated by Zona Modeler from the definitions within the **`TileTestSchema.xml`** model file. There are several key user-defined properties used to model system behavior:

Table 28-1. Tile Test Entity

Property	Description
System.cheatCorrection	Defines the number of times the GSS detects a player cheat.
Private.lastLogin	Defines the date of the last login.
Public.x	Defines the X position variable.
Public.y	Defines the Y position variable.
Public.color	Defines the Tile color.
Public.blink	Defines the Tile blink state.

This chapter will focus on the Server-side aspects that are not covered in *Simple Client Creation on page 169*.

Programming the Client

In these code snippets, we show the various parts of the client implementation not previously shown in the TrackerClient implementation.

Creating a Character

Creation of the character is a simple process focusing on the call to ZonaServices to create the actual Character. Note that the Character type was set to **`ZONA_ET_DEFAULT`** in the header definition. Other type exist for NPCs and other roles within the game. These and other constants are defined in this file:

```
%ZONA_HOME%\include\zaf\ZonaGlobalConstants.h
```

Character creation happens in this file:

```
%ZONA_HOME%\samples\TileTest\client\ZonaNet.cpp
```

If no Character exists, then create one. Otherwise, use the Character from a call to **`ZonaServices::getCharacters()`**:

```
if (characters.getSize() == 0)
{
    // Create a new character
    wchar_t wname[128];
    wcscpy(wname, username);
    wcscat(wname, L"_char");
    m_character = new TileTestClientEntity();
    m_character->setName(dlgUsername, strlen(dlgUsername));
    ((TileTestClientEntity*)m_character)->setX(-1);
    ((TileTestClientEntity*)m_character)->setY(-1);
    ((TileTestClientEntity*)m_character)->setBlink(1);
    ((TileTestClientEntity*)m_character)->setColor(color);
    m_zonaServices->createCharacter(*m_character);
    ASSERT(m_character->getEntityId() != 0);
} else
    m_character = characters[0];
```

The **`TileTestClientEntity`** class is autogenerated by Zona Modeler from the Entity model definitions contained in the **`TileTestSchema.xml`** Zona Model file.

Modifying the Character

To modify the Character, you must modify the Character Entity Properties and then publish these "dirty" Properties back to the Managing GSS for validation. Following successful validation, the GSS will update the Character information across the cluster.

The main on-screen Character movement function is found here:

`%ZONA_HOME%\samples\TileTest\client\TileTestDlg.cpp`

The **`CTileTestDlg::movePlayer()`** function updates the local Character display using the **`TileTestClientEntity::setPos()`** utility function.

```
bool CTileTestDlg::movePlayer(int offsetX, int offsetY)
{
    TileTestClientEntity *e= ZonaNet::getLocalEntity();
    // test map boundary
    if ((e->getX()+offsetX) >= MAP_WIDTH ||
      (e->getY()+offsetY) >= MAP_HEIGHT ||
      (e->getX()+offsetX) < 0 ||
      (e->getY()+offsetY) < 0)
      return false;
    int posX = e->getX() + offsetX;
    int posY = e->getY() + offsetY;
    // test tile
    if (m_clientColl && (m_map.isPassable(posX, posY) ==
    false || ZonaNet::isTileFree(posX, posY) == false))
      return false;
    e->setPos(posX, posY);
    return true;
}
```

The main Character Property data update function is found here:

`%ZONA_HOME%\samples\TileTest\client\TileTestClientEntity.cpp`

The **`TileTestClientEntity::setPos()`** utility function uses the Zona Modeler-generated **`setX()`** and **`setY()`** functions to update the local Character Properties:

```
void TileTestClientEntity::setPos(int x, int y)
{
    setX(x);
    setY(y);
}
```

The main Character data update publishing function (to alert the GSS) is found here:

```
%ZONA_HOME%\samples\TileTest\client\ZonaNet.cpp
```

The **`ZonaNet::updateToServer()`** function alerts the GSS that the Client is requesting a change in the global Character Properties, and triggers Server-side validation:

```
void ZonaNet::updateToServer()
{
    m_zonaServices->getCharacter()->publish();
}
```

Listening for Server Updates

The two main functions for receiving GSS updates to the Client Entities are coded within this file:

```
%ZONA_HOME%\samples\TileTest\client\ZonaTileTest.cpp
```

The **`ZonaTileTest::onNotifyEntityPropertyUpdate()`** function handles the reception of data updates from the GSS.

```
void ZonaTileTest::onNotifyEntityPropertyUpdate
                                    (ZonaClientEntity* entity)
{
    printf("ZonaTileTest::onNotifyEntityPropertyUpdate\n");
    TileTestClientEntity* myEntity =
                                    ZonaNet::getLocalEntity();
    TileTestClientEntity* e = (TileTestClientEntity*)entity;
    if ( !e )
        return;
    int a = entity->getEntityId() ;
    int b = myEntity->getEntityId();
        bool isXORYChanged =false;
    bool isLastLogin = false;
    BitArray *bm = e->getDirtyPropertyBitMask();
    if (bm!= NULL && ((bm->bitValue(e->getXBitIndex()) != 0)
              || ( bm->bitValue(e->getYBitIndex()) != 0 ) ))
      isXORYChanged = true;
    else
      isXORYChanged = false;
    if
(bm!= NULL && (bm->bitValue(e>getLastLoginBitIndex()) != 0))
      isLastLogin = true;
    else
      isLastLogin = false;
    if ( entity->getEntityId() == myEntity->getEntityId()) {
        if ( isXORYChanged )
          PlaySound
            ("bang.wav", AfxGetInstanceHandle(), SND_ASYNC);
        if ( isLastLogin ) {
          char pdata[200];
          int  length =0;
          char *data = e->getLastLogin(length);
          CString myString;
          m_dlg->GetWindowText(myString);
          sprintf
```

```
                (pdata,"%s (Login Since:%s)",myString, data);
            m_dlg->SetWindowText(pdata);
            delete[] data;
          }
      }
      int x = e->getX();
      int y = e->getY();
}
```

Check the code comments in the **`ZonaTileTest.cpp`** file for a description of the optimizing bitmask algorithm used in this function.

The server-side GSS function that validates these Client updates is **`ZonaCharacterValidate::onValidateEntityPropertyUpdate()`**.

Managing the Client-Side Entities

You do not have to write code to explicitly manage multiple server-side Entities because the GSS's transparent Zona Entity Manager (ZEM) handles this administrative task for you, freeing you to concentrate on coding game logic. All you need to do is get a pointer to the list of all Entities currently being managed by ZEM.

Entities are added to the ZEM immediately before the Client executes the **`EntityCallback::onNotifyEntityJoinedSphere()`** callback, and are deleted from the ZEM immediately after returning from a **`EntityCallback::onNotifyEntityDepartedSphere()`** callback. Do not attempt to explicitly delete Entities while they are being managed by the ZEM.

To retrieve all the Entity objects currently being managed by the ZEM, use the **`ZonaServices::getAllEntities()`** function. This will return a pointer to the Entity object vector **`ZonaBaseEntityPtrVector`**.

The Client-side code that retrieves the Entities from the ZEM is found in this file:

```
%ZONA_HOME%\samples\TileTest\client\ZonaNet.cpp
```

The **`ZonaNet::isTileFree()`** function retrieves the managed Entities when it is referenced by the **`TileTestDlg::movePlayer()`** UI handler for displaying legal Tile moves:

```
bool ZonaNet::isTileFree(int x, int y)
{
    ZonaBaseEntityPtrVector entities;
    int entityCount = m_zonaServices-
    >getAllEntities(&entities);
    TileTestClientEntity *e;
    for (int i=0; i<entityCount; i++)
    {
      e = (TileTestClientEntity*)entities[i];
      if (e->positionIs(x,y))
          return false;
    }
    return true;
}
```

The **`ZonaServices.getAllEntities()`** function fetches from the ZEM a list of all local Entities (and all remote Entities) within the Character's SOI and returns the number of Entities as an **`int`**. This function executes locally and does not access the GSS.

This contrasts with the **`ZonaClientEntity.getChildEntities()`** function. This bypasses the Client-side ZEM to fetch from the GSS a list of all Child Entities owned by the calling Entity and returns an **`int`** error code indicating the success or failure of the operation.

Programming the Server

This section presents the Server-side GSS Plugin code that validates the Client-side code. There are several key files used for server-side Entity management and validation:

`%ZONA_HOME%\samples\TileTest\ServerPlugIn\TileTestServerPlugin.cpp`

Defines the Server-side Entity object behavior. Contains two behavior validation and constraint functions. These are the **`TileTestServerPlugin::isTileFree()`** and **`TileTestServerPlugin::oneTileMove()`** functions.

`%ZONA_HOME%\samples\TileTest\ServerPlugIn\ZonaServer\*.cpp`

This directory contains seven implementation files:

```
ZonaCharacterValidate.cpp
ZonaEntityValidate.cpp
ZonaGameStateValidate.cpp
ZonaGuildValidate.cpp
ZonaRegionValidate.cpp
ZonaSystem.cpp
ZonaTimerEvents.cpp
```

These implementations handle game initialization, Client data validation, and data publishing and synchronization between other GSSs and their Clients.

Managing the Server-side Entities

You do not have to write code to explicitly manage multiple server-side Entities because the GSS's transparent Zona Entity Manager (ZEM) handles this administrative task for you, freeing you to concentrate on coding game logic. All you need to do is get a pointer to the list of all Entities currently being managed by ZEM.

Entities are added to the ZEM immediately before the Client executes the **`EntityCallback::onNotifyEntityJoinedSphere()`** callback, and are deleted from the ZEM immediately after returning from a **`EntityCallback::onNotifyEntityDepartedSphere()`** callback. Do not attempt to explicitly delete Entities while they are being managed by the ZEM.

To retrieve all the Entity objects currently being managed by the ZEM, use the **`ZonaGSPublish::getAllEntities()`** function. This will return a pointer to the Entity object vector **`ZonaBaseEntityPtrVector`**.

The Server-side code that retrieves the Entities from the ZEM is found in this file:

`%ZONA_HOME%\samples\TileTest\ServerPlugIn\`

TileTest_ServerPlugin.cpp

The **TileTest_ServerPlugin::isTileFree()** retrieves the managed Entities when it is referenced by the Entity validation function **ZonaEntityValidate::onValidateEntityPropertyUpdate()** to check for legal Tile move behavior:

```
bool isTileFree(int myId, int x, int y)
{
    ZonaBaseEntityPtrVector entities;
    int entityCount = getAllEntities(&entities);
    TileTestEntity *e;
        int eX, eY;
    for (int i=0; i<entityCount; i++)
    {
      e = (TileTestEntity*)entities[i];
      eX = e->getX();
      eY = e->getY();
      if (myId!= e->getEntityId() && (x == eX && y == eY))
          return false;
    }
    return true;
}
```

Validating the Client Request

The incoming Client property modification requests are received by the **`ZonaEntityValidate::onValidateEntityPropertyUpdate()`** function:

```
bool onValidateEntityPropertyUpdate
          (ZonaServerEntity* entity, ZonaServerEntity* prev)
{
    entityState state;
    TileTestEntity* e = (TileTestEntity*)entity;
    TileTestEntity* last = (TileTestEntity*)prev;
    state.posX = e->getX();
    state.posY = e->getY();
    zprintf("onValidateEntityPropertyUpdate() entityId=%d
sent x=%d y=%d lastX=%d lastY=%d\n",
                e->getEntityId(),
                state.posX, state.posY,
                last->getX(), last->getY());
    // ASSUMPTION: First location sent is valid
    if(last->getX() == -1 || last->getY() == -1) {
        return true;
    }
    if (isTileFree(e->getEntityId(),
                        int(state.posX), int(state.posY))
        && map.isPassable(int(state.posX), int(state.posY))
        && oneTileMove(e, last) ) {
        zprintf("%i made a valid move\n", e->getEntityId());
        // Player position is good
        return true;
    }
    else {
        e->setX(last->getX());
        e->setY(last->getY());
        e->setCheatCorrection(last->getCheatCorrection()+1);
        // save to DB
        e->save(false);
        zprintf
           ("%i made an INVALID move!\n", e->getEntityId());
        return true;
    }
}
```

After checking the validity of the nature of the property modification requested, the data is processed for game-specific logical validation. The **`getCheatCorrection()`** and **`setCheatCorrection()`** functions are autogenerated by Zona Modeler from the definitions within the **`TileTestSchema.xml`** model file.

When a property change is validated, the Plugin sets that Property dirty. This will cause Terazona to push this changed property value to all subscribed GSSs and Clients. If the validation function returns false, then the GSS publishes propagates the older, correct Character state back to managed Client. This corrected data is received by the Client's **`ZonaTileTest::onNotifyEntityPropertyUpdate()`** function and its Entity properties are reset to their correct values.

Managing the Regions

The Regions are also managed within this file:

```
%ZONA_HOME%\samples\TileTest\ServerPlugIn\ZonaServer\
                                        ZonaRegionValidate.cpp
```

This defines a very simple, rectilinear grid Region structure.

```
byte g_regionType = 50;
static byte g_currentRegionId = 0;
static int g_numberOfRegions = 1;
int onGetRegionCount(int mapId)
{
    return g_numberOfRegions;
}
void onInitializeRegions(){}
int* onGetAllRegionIds(unsigned int *numIds)
{
    int* allRegionIds = NULL;
    allRegionIds = new int[1];
    allRegionIds[0] = g_currentRegionId;
    *numIds = 1;
    return allRegionIds;
}
int* onGetRegionNeighbors(int mapId, int regionId, byte
regionType, unsigned int *numIds)
{
    *numIds = 0;
    int *allRegionIds = NULL;
    return allRegionIds;
}
bool onPlaceEntityInRegion(ZonaServerEntity *entity, int
```

```
mapId, int regionId)
{
    zprintf("\nonPlaceEntityInRegion\n");
    TileTestEntity *e = (TileTestEntity *)entity;
    char loginTime[50];
    SYSTEMTIME mySYSTIME;
    GetLocalTime(&mySYSTIME);
    sprintf(loginTime,"%i/%i-%i:%i",mySYSTIME.wMonth,
                                    mySYSTIME.wDay,
                                    mySYSTIME.wHour,
                                    mySYSTIME.wMinute);
    zprintf("login Time : %s\n",loginTime);
    e->setLastLogin(loginTime,strlen(loginTime));
    e->save(false);
     return true;
}
```

The **`onPlaceEntityInRegion()`** function uses the **`setLastLogin()`** function autogenerated by Zona Modeler from the definitions within the **`TileTestSchema.xml`** model file.

Glossary

GSS. Game State Server

Sphere Server. Server that manages the load balancing of the Game State Server

Dispatcher. Handles login and hand off of players to the Game State Servers

ZAC. Zona Admin Control

GGS. Handles all text communication between clients that are members of Guilds

Messaging Server. This handles all of the low level messaging between the servers and the clients

Zona Admin. The GUI for administrative control of the Terazona servers

JMS. Java Messaging Service

JDBC. Java Database Connectivity

Character Entity Properties. The record containing all of the character data that is stored in the database

Regions. Developer defined logical representation of areas within the game.