System Design Document

ARENA Project

February 12, 2003

Contents

List of Figures

1 Introduction

1.1 Document Overview

This System Design Document (SDD) presents the technical details of the ARENA system design. More information about the specific features and the motivation for ARENA can be found in the Requirements Analysis Document (RAD) and the Problem Statement.

This document starts with an introduction to the architecture and the design goals to be considered. Then it presents the proposed system architecture by describing the subsystem decomposition and the subsystem services, defining the hardware/software mapping and explaining the management of persistent data. Access control and security issues are addressed. The global software control and boundary controls are described. Finally a glossary of important terms is provided.

1.2 Design goals

- Employment of the FRAG framework:
  The FRAG framework provides distributed object management and state replication between peers. To develop a game, only the creation of objects and algorithms is necessary. The framework is work-in-progress and is supposed be modified and extended.
- Algorithmically defined game world:
  The world in which the game takes place is created by a randomized algorithm, so two instances of the game are always different.
- Restricted access to games:
  Players can define buddies - other persons, who are allowed to join the game. These players are stored in the buddy list and play on the same team as the initiator, if the game is set to team play mode.
- Game setup without network configuration:
  On startup, the system detects running games in the LAN and offers to connect to one of them or start a new game. The user doesn't have to set up network parameters like protocols, addresses or ports. Not even the initiator of a game needs to set game-specific options other than an unique name for his/her game. The Rendezvous architecture offers zero-configuration network service discovery and usage, thereby eliminating the need to develop a proprietary game detection/announcement protocol.

- No connection to a server:
  Even people without connection to the Internet or another wide area network can play the game (over Wireless LAN, Bluetooth, or any other local network).

- Platform-independent, open standard-based game design:
  The system is platform-independent to allow usage on a wide range of devices, even Java-enabled handheld devices as soon as they become powerful enough. Therefore, all code is written in Java, with the possible exception of I/O-driver and lower-level network components, such as the Mac OS X Rendezvous integration.

- Integration of a new I/O device:
  Research in Augmented Reality, Human Computer Interaction, and related fields produced new devices like data gloves, head trackers or 3D glasses. The game uses at least one new input device,in this case a gyroscope with a head mounted display. It is also considered to make use of speech recognition.

- Exhibition of interactive response time:
  The game is fast enough to enable realtime playing (at least 12 frames/sec) on the provided iBooks over Wireless LAN.

- Robustness:
  The game should be stable and playable even when losing network connection.

- Modifiability:
  It must be easy to change and extend the components of the game, e.g. the GUI.

2 Proposed System

2.1 Overview

This section describes the requirements of the distinct proposed software architecture. We describe the subsystem decomposition in the way we designed and planned to implement it. This includes the decomposition as well as the proposed services and their interaction. These decisions are based on the RAD (Requirement Analysis Document) and the PS (Problem Statement). We also describe our design decisions and reasons for the hardware/software mapping, the consistent data management, the access control handling, the global software control and the boundary conditions.

2.2 Subsystem decomposition

The system is divided into nine subsystems, which are described in terms of their services.

2.2.1 Subsystem Decomposition

The following diagram shows the system decomposition and the included subsystems:

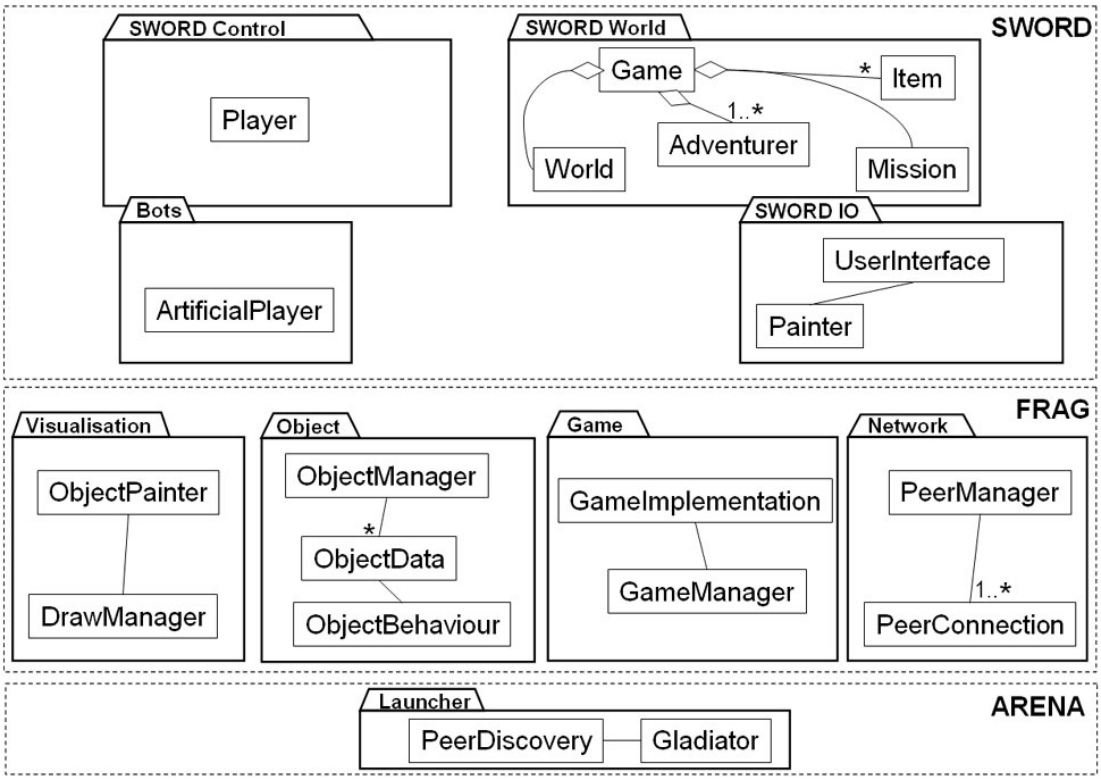

Figure 2.1: Subsystem Decomposition

Launcher	The Launcher is the initiating subsystem of SWORD.
SWORD Control	The SWORD subsystem is responsible for controlling the specific (concrete) object instances of the game.
SWORD World	The game world subsystem is responsible for creating the game world; using a fractal algorithm, it computes the world which corresponds to the specific mission. It initiates the game with all necessary characters and items and places those into the algorithmically created game world.
Bots	The Bots (formerly AC or ArtificialControl) subsystem is responsible for instantiating the Artificial Players of the game; these are the opponents of the Human Players - the real players in the real world; the Artificial Players are controlled by an algorithm.
SWORD IO	The SWORD IO subsystem is responsible for providing a visual interface used by the Human Player to navigate his Adventurer (character) in the game.
Game	The Game subsystem is responsible for initiating and joining games and for handling all related management issues.
Object	The Object subsystem is responsible for the objects of the game world; especially sending and getting their state update to, respectively from the network component.
Network	The Network subsystem is responsible for exchanging connection data between the local peer and other peers on the network.
Visualisation	The Visualisation subsystem is responsible for drawing all visual objects of the game on the screen.

2.2.2 Subsystem Services

This section describes the services each subsystem provides along with their prerequisites and output products.

2.2.2.1 SWORD Control Subsystem

Player	The Player is the common interface used for navigating the Adventurer in the game world by the Human Player from the real world. This interface is also responsible for management commands, e.g. loading and saving games and the communication between the players.

2.2.2.2 SWORD World Subsystem

Game	A Game represents a SWORD game instance, which is created after the start of SWORD and is inheritated from GameImplementation; this class instantiates the game with its algorithmically created world and the mission chosen; it contains all items and adventurers of the game.
interface of subsystem	*initGame:* This service initializes the SWORD-game. Especially the world, mission and items. The parameter is GameManager.
Adventurer	An Adventurer represents all characters that are in SWORD (e.g. actors, monsters, ...) and is the boundary object to real player and also to the Artificial Players. The basic actions an adventurer can perform are:
interface of subsystem	*useItemWith:* The item is used with another item. *dropItem:* The item is dropped to the floor and attached, and is therefore visible for all again. *move:* The position of the adventurer is changed to the new position. *lookAt:* The details of an item are shown.
World	A World is an instance of a chosen game mission, which is created right after the SWORD game has started. Every game world has its own algorithmically designed and unique environment.
Mission	A Mission represents a scenario of the game and the steps, which the players should accomplish to win the game.
Item	All not moving objects are items with the basic task useItem. The information about the item is stored in its properties. An item can be a SingleItem or an ItemContainer which can store ItemContainers or SingleItems.

2.2.2.3 Bots Subsystem

ArtificialPlayer	The Artificial Player is algorithmically created service which navigates the adventurers not controlled by the human players in the real world. Those adventurers in the game world could be opponents or helpers of the adventurers, navigated by the human players.

2.2.2.4 SWORD IO Subsystem

UserInterface	The user interface distributes commands, given from the player in the real world which navigate the Adventurer in the game world. This is the main interface between the real world and the game. The UI's task is to interact with the user by displaying information provided by other subsystems and by fetching user input and altering the data, if necessary. The User Interface will provide a set of user-friendly windows, e.g. login, the main game GUI and status messages. In turn, the UI subsystem is responsible for getting user input from the SWORD subsystem and delegating those events to the Game subsystem and the Object subsystem. The user inputs are voice, keyboard, mouse and headtracker commands. The GUI consists of one full screen window which is divided into three main areas. On top there is the area where status information about the character such as strength and mana is provided. Additionally there is a menu where the human player can choose to exit, save and pause the game, to communicate with other players and to set the options of his screen. The biggest area in the middle of the screen provides the game with the world surrounding the character, but only in the direction the character is facing to. Here the objects of the world like items and other players are displayed to the player. The bottom area of the screen itself is divided into three sections which display the item container, the abilities of the character and the map. The item container shows the items the character has collected during the game. If the character drops or loses an item it is removed from the item container and thus no more displayed to the player as an item he possesses. The area in the middle displays the abilities the character has achieved during the game, such as magic spells. On the right side there is the map which displays a bird perspective of the world surrounding the character including hills, rivers, streets, buildings and villages.
Painter	The Painter class manages the visualization of the current GUI and the individual message windows. It also uses the draw methods provided by the DrawManager in the Visualization subsystem.
interface of subsystem	*showWindow*: Puts a message window on the screen. The parameters the window's name and the message itself. *setGUISkin*: Provides a method to change the GUI skin. Parameter is the name of the skin file.

2.2.2.5 Game Subsystem

GameManager	The GameManager class handles the creation and joining of games. It provides the interface for interaction with the underlying layer (Network components or ARENA itself). The actual work of creating a game with objects etc. is delegated to a subclass of GameImplementation.
interface of subsystem	*joinGame*: This method provides the functionality of getting into a running game. The parameters are the names, the IP addresses and the ports. *initGame*: makes a forward call to the game implementation in which the logic of the game is specified. *newPeer*: adds one new peer to the PeerManager. The parameters are the name, the IP address and the port. *setImplemention*: sets the GameImplementation to the gameManager
GameImplementation	The GameImplementation class implements a concrete FRAG-based game, in this case the SWORD game.
interface of subsystem	*gameJoined*: to join to a existing game using the gameManager as parameter. *initGame*: initialize the game with the game manager as parameter

2.2.2.6 Object Subsystem

ObjectManager	The ObjectManager class manages all the objects in the game and distributes the control of an object among the peers. The ObjectManager class sends state updates for its objects to the other peers.
interface of subsystem	*storeObject*: Puts the objectdata into the ObjectTable. Stored is the object's name and objectdata. *processEvent*: Process the specified event concerning to the object class type. The event types are: ObjectUpdateEvent, ObjectControllerEvent, ObjectKilledEvent and ObjectActionEvent. *checkModified*: checks if the objects has been modified. The parameter is the name of the object. *sendObjectUpdate*: creates an new ObjectUpdateEvent which is handled in the processEvent method. Parameter is a ObjectData object. *acquireControl*: puts the control of the object to the player of this instance of the game. Parameter is the object name. *releaseControl*: releases the control of the object to the player of this instance of the game. Parameter is the object name. *checkAllFocuses*: checks for all object the focus property. *objectMoved*: is called when a object is moved. Parameter is the object name. *releaseAndKillObjects*: delete the object from the object table and calls checkAllFocuses. Parameter is the current owner of the object. *objects*: Returns a hashtable which consists all of the objects in the objectmanager. No parameters *importObjects*: Imports a hashtable of objects into the object table. Parameter is a hashtable.
ObjectData	The ObjectData class handles the state of a single object, e.g. position. Most objects (Item, Adventurer, ..) from the SWORD subsystem have a associated ObjectData class.
ObjectBehavior	The ObjectBehavior class manages the behavior of an object and is associated from the ObjectData class.

2.2.2.7 Network Subsystem

PeerManager	The PeerManager class manages the peer connections for the local peer and deals with any unexpected loss of connections. Therefore it provides a service for addingPeers with the parameters: IP address and player name. The discovery of new peers is done by the PeerDiscovery component.
interface of subsystem	*peerDead*: creates a PeerDeadEvent and removes the peer from the peer manager. Parameter is the PeerConnection. *addPeer*: Adds the peer to the peerManager with the IP address and the port number. Parameter is the name, the IP address and the port number.
PeerConnection	The PeerConnection class represents a single connection to another peer, including the corresponding socket, in- and output streams.
interface of subsystem	*establishConnection*: Establish a connection to a peer manager. Parameters are peermanager, the ip address and the port number. *processEvent*: Sends a event over the network. Parameter is the event.

2.2.2.8 Visualisation Subsystem

DrawManager	The DrawManager distributes common draw methods for visualization and has a method, that calls the visualization-method of the ObjectPainter interface. The DrawManager gets the current GameManager and current graphic container as parameter.
interface of subsystem	*drawAll*: This method provides functionality for calling the draw method on each object which it gets from the ObjectManager.
ObjectPainter	The ObjectPainter Interface provides an interface for the visualisation of single objects. The ObjectPainter implementation manages the allocation of the visualisation resources and passes them to the assigned DrawManager visualisation method.
interface of subsystem	*draw*: This method makes a forward call to the DrawManager where the real drawing things are done so that the hole drawing things aren't transmitted over the network. The parameters are the ObjectData and the DrawManager.

2.2.2.9 Launcher

Gladiator	This is the main application to run a game. Gladiator initializes PeerDiscovery and requests a list of running games and peers. Starts the game by calling the GameManager and passing the parameters: IP addresses and player names.
PeerDiscovery	Discovers running games and peers.

2.2.3 Interaction of Subsystem Services

This section shows the interaction among the subsystems resulting from the user tasks described in the RAD.

When creating a game a service is called which is responsible for the initialization of the world, mission and items.
Players who want to join are detected by PeerDiscovery which calls a service joinGame in GameManager with the parameters IP address, port and the name of the player. The GameManager delegates this to the PeerManager which creates a new PeerConnection. As described in figure 2.2. To visualize the game the DrawManager gets the objects that should be drawn on the local screen from the ObjectManager. Therefore the getObjects service in ObjectManager is called which returns an enumeration containing ObjectData objects. These objects are drawn by the ObjectPainter.

The interaction between real and game world is done by the UserInterface. Any input is given to Player which calls the basic services in Adventurer for manipulating itself: move, attack, pickupItem, dropItem, lookAtItem and useItemWith.
Mostly the same is done by the ArtificialPlayer apart from generating the input algorithmically.

When an Item or Adventurer changes its position the ObjectManager is notified and creates an event consisting of important data, e.g. the new position. This event is broadcasted to the PeerConnections. For clarification see the overview figure 2.3.

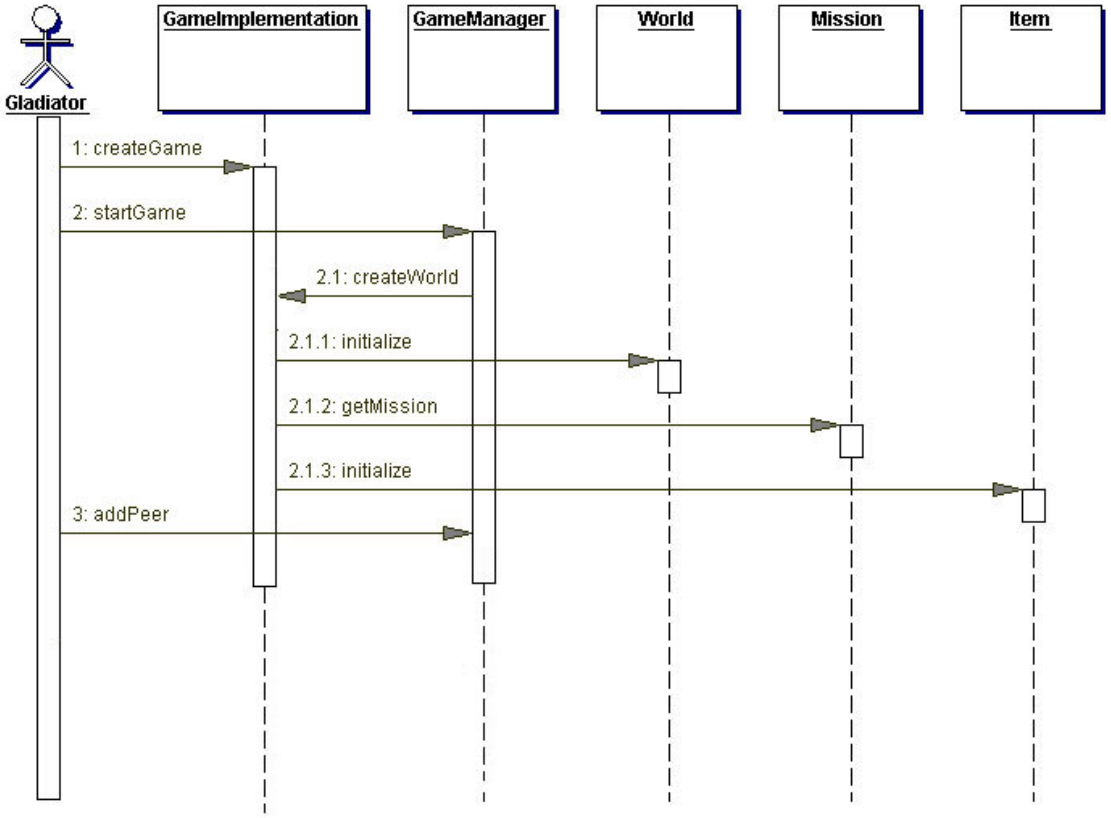

Figure 2.2: Subsystem eventflow by starting the game

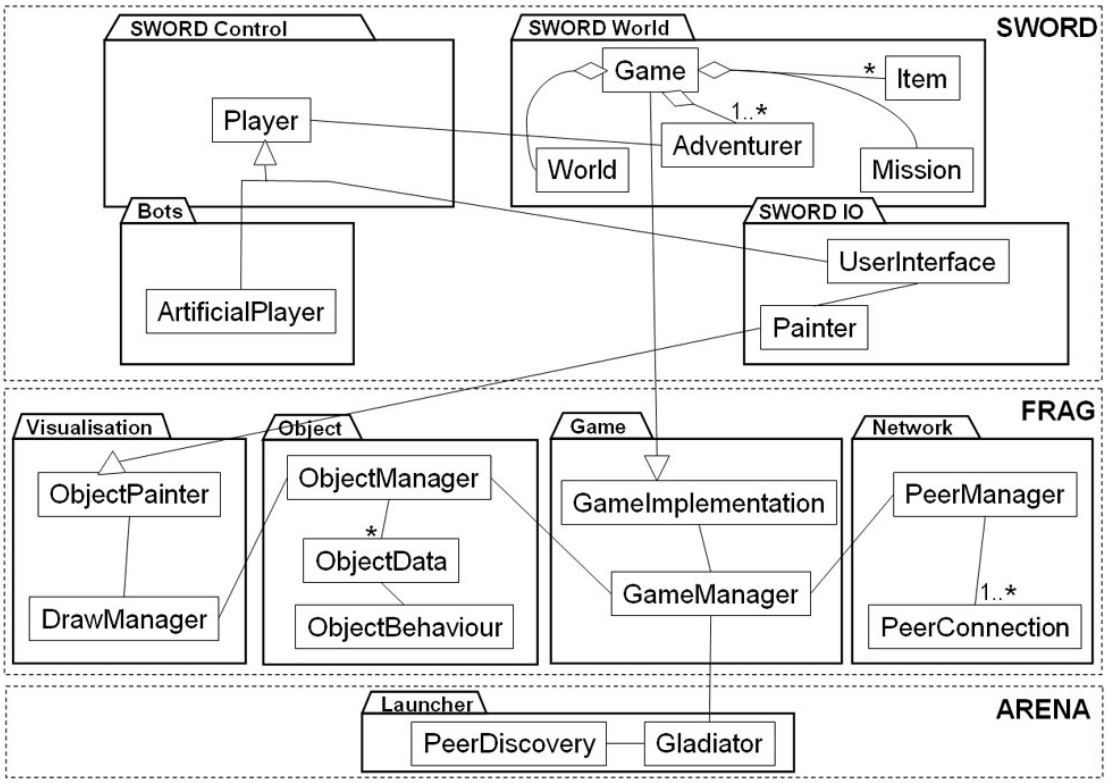

Figure 2.3: Subsystem interaction

2.3 Hardware/Software Mapping

Based on the task and client feedback our system consists of one or more laptops running MacOS X. Development is thus done based on this platform.

Figure 2.4 gives a general overview of the main architecture and their relationship to SWORD.

Network communication is managed by FRAG using a TCP/IP based protocol. Java was chosen as a main programming language to achieve the goal of compatibility with a future usage of the framework. However some of the I/O drivers will be written in C++.

The target environment consists of the following parts:

1. Laptop: Apple iBook, Motorola G3 processor, running MacOS X
2. Head Mounted Display
3. Inertial Tracker Intersense InterTrax 2

Figure 2.3 is a summary of the mapping between hardware components and the software environment.

The SWORD game prototype will be playable but restricted to simplified graphics and gameplay due to time restrictions. However, the FRAG framework will be fully functional, and the game can be used as a prototype for future development.

2.3.1 General System Performance

Due to the game nature of SWORD it is necessary that we have fast response times for the information displayed to the user.

For any network request SWORD aims at a maximum response time of just a few seconds. However, SWORD cannot guarantee a specific response time, due to circumstances beyond our control like heavy network traffic or wireless LAN interference.

The general performance also depends on the implementation of the FRAG infrastructure, since most interactions between subsystems as well as between the system and the user are based on the FRAG services.

2.3.2 Input/Output Performance

SWORD is expected to provide at least 12 frames per second on the output device.

2.3.3 Processor and Memory Allocation

All computation is being done by the 700 Mhz G3 processors of the laptops and by their graphic chips. However, if multiple laptops are used, different parts of the world are to be computed by different units. As well, in such a case, object control is also distributed between laptops. Concerning the memory allocation it's always good to have enough RAM installed so the computer doesn't have to outsource data on the HD. We use iBooks with 640 MB of RAM.

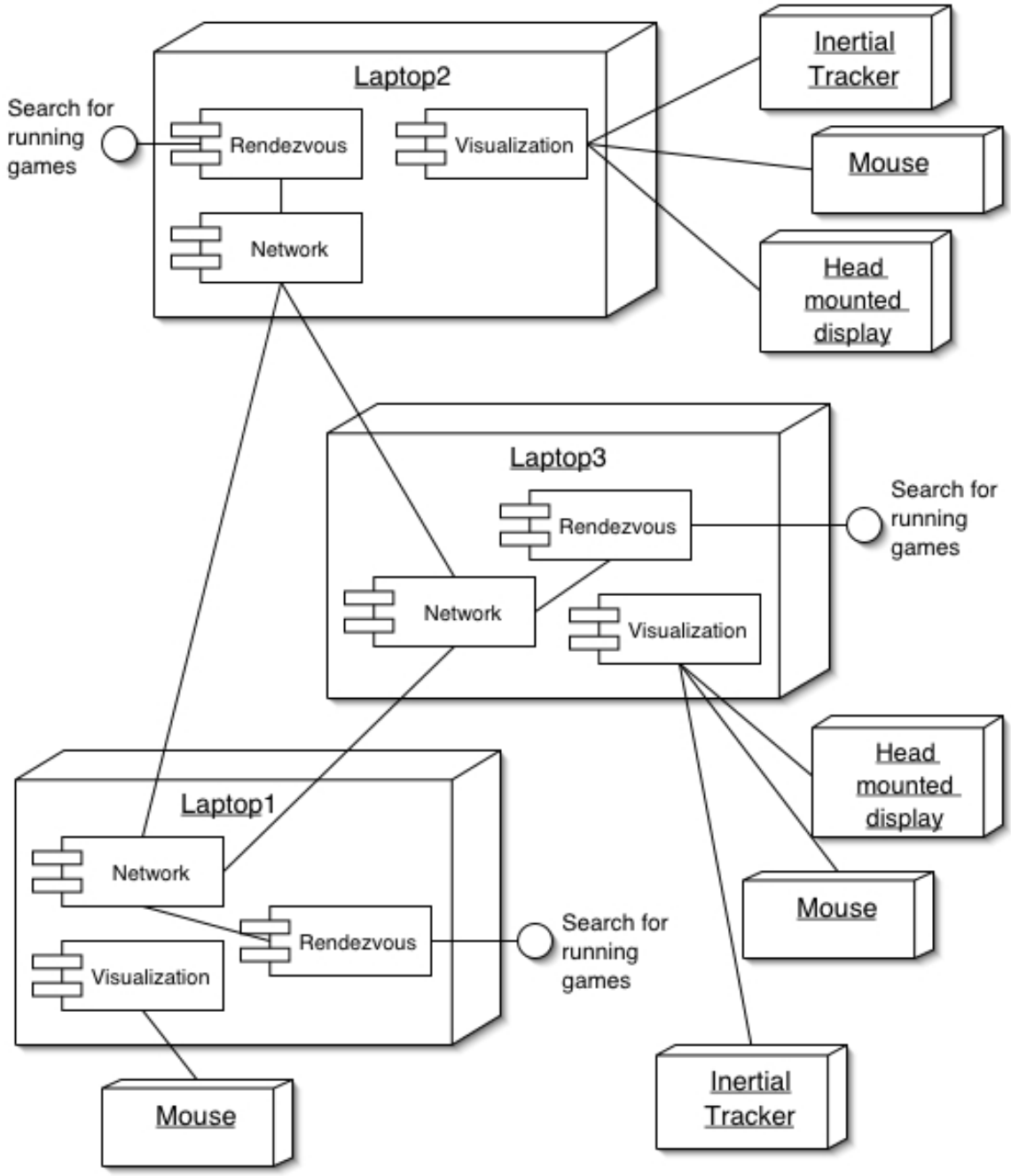

Figure 2.4: Allocation of SWORD subsystems to hardware

Component	Interface	Operating System	Connected to
Laptop	USB, FireWire, VGA, WaveLAN	Mac OS X	—
HMD	VGA	—	laptop
Inertial Tracker	Serial via USB adapter	—	laptop

Figure 2.5: Interfaces and Operating Systems used by SWORD hardware

2.3.4 Connectivity and Network Infrastructure

The laptops are connected with 11 Mbit wireless LAN or 10/100 Mbit Ethernet.

The connectivity is done via FRAG which provides a peer-to-peer architecture using a TCP/IP based protocol. TCP/IP allows reliable data transfer as long as the connection remains active. A wireless connection can fail if the components move out of range or if extensive interference consistently weakens or garbles the signal.

The anticipated bandwidth requirements should be in the order of magnitude of video streams, so several hundred Kb/sec would result in an acceptable user-system interaction performance. However, plans to integrate speech recognition and modern I/O devices may require higher bandwidth or might delay the overall response as the CPU has more data to handle.

Since the system uses pro-active routing (each peer maintains a connetion to every other peer), the bandwith expense grows quadratically ($\frac{n(n-1)}{2}$ connections for n peers). For that reason, the prototype might not be able to run according to the requirements with more than about 5 peers on WLAN and 15 peers on 100 Mbit Ethernet, respectively.

2.3.5 Modern input device

After considering multiple modern devices we had available, we decided to use a head mounted display with head motion tracker and speech recognition as the modern input stated in the requirements. The head mounted display doesn't require special drivers as it uses the standard VGA output of the laptop. The head motion tracker however does require a driver. It will be implemented as independent driver program, written in C++, and connected to the SWORD with the TCP/IP protocol inside one system locally or alternatively, remote. Remote connection might be useful for performance issues, when another laptop can calculate the input of the device. However, it should be possible to run it all on one system.

Due to the 2D nature of first edition of SWORD, its likely that the head motion detector will have following control features for the system:

- Axis 1 (Yaw) - not used or implemented in the system later.
- Axis 2 (Pitch) - Head forward makes the adventurer running forward, and head backwards makes the adventurer moving backwards. It will be most likely a digital (move/not move) condition, speed independent on the angle of head.
- Axis 3 (Roll) - Head left/right will make the adventurer rotate left and right. Rotation speed will be most likely done in a few possible grades. (triggered by 5, 10, 15 degrees

angles). However, it might be found reasonable to implement this control as simple two state signal, too, according to the Adventurer class interface.

Another input device is the speech recognition. It is based on the JavaSpeechFramework and will be written in Java. We will use this to control the main functions in SWORD (for example: save game, resume to game, leave game and another) These actions will be controlled by speech recognition when you use the head mounted display because you can't find a key on the keyboard then. You can also use the speech recognition when you don't use the head mounted display as alternative input device to keyboard or mouse.

2.4 Persistent Data Management

Several properties of a game can be saved to a file system and loaded at a later point in time. The goal is to allow a player to suspend and resume a game in a — for both the user and the system developer — simple and effective way that avoids inconsistencies or usability issues. The data describing the game world and the data describing an adventurer are saved independently.

2.4.1 World Data

The world consists of the generated map and the items located on the map, including items that are owned by adventurers. If a player leaves the game he is asked if he wants to save the World Data. The data that is actually saved contains the seed key for the World Generation Algorithm and all the items in the game with their current (at the time of saving) position. The adventurers themselves are not saved as part of the world.

Saving the World Data is optional on leaving the game; loading the World data is optional on starting as an alternative to Create New Game and Enter Game.

2.4.2 Adventurer Data

The properties (abilities, strength, life power, magic power) of an adventurer can be saved and re-loaded at a later time and also for different game worlds and missions.

Moreover, the position data of an adventurer is saved on exiting, allowing a player to resume a game at a later point in time starting at the same position. When a player exits the game, all items that were in the inventory of the adventurer reappear on the map, about at the position where the adventurer left the game.

Saving the Adventurer Data data is done automatically on leaving the game. It is not possible to revert to an earlier status during or after the game. Loading the Adventurer Data is also done automatically at startup; there is also an option to create a new adventurer character

2.4.3 Data Management design rationale

This approach has two main advantages. First, it allows a player with a mission-critical item to leave a game without blocking the other players from finishing the mission. Moreover, when the player resumes or re-joins the game, he can instantly regain all items he had in his inventory when he left the game (of course only if not another Adventurer in the game has already picked up the items from this place).

This concept also avoids inconsistency since a player who has suspended a game and wants to continue the game always has the possibility to join the game, which has been relaunched by some other player or still in progress, or himself relaunch the game in the state when he has left it. One the other hand merging two relaunched games is impossible.

Saving World Data and Adventurer Data indepentdently allows the player to develop his Adventurer over several games. This gives the game a role-playing character and fits fine with the concept of property-defined characters types.

Making the Adventurer Data auto-saved and not recoverable makes it important for the player to take care of his character and emphasizes the role-playing aspect. A player who loses his character has to start over again.

2.5 Access control and security

In a MOG (Multiplayer Online Game) with a dedicated server the operator of the server usually has the privilege to restrict the access to a running game. Moreover, by quitting the game on the server machine, he/she forces the other players to leave that particular game. In a peer-to-peer world running a distributed application does not depend on a single machine anymore, but the need of access control remains. Since all peers are equal, each one must have the same control rights. Therefore the access to the game is restricted by a password, which is set by the initiator of the game. The password is sent to all peers present upon initiating. This is the only initiator-specific task regarding access control. If a person wishes to join on a later stage, he/she can contact any player and obtain the password via chat mode or e-mail. Following alternatives of the password-based access control were discussed:
- asymmetric cryptographic process deploying public/private key pair. This alternative is too difficult to be implemented on time.
- no access control at all. This alternative is easy to implement but conflicts with the PS.

2.6 Global software control

The following section will describe the global software control flow that means the sequencing of actions in our system. There are three types of control flow paradigms: procedure-driven, event driven or threaded control flow paradigm. In our multiple System control, it is possible to select different control paradigms for different components.

When selecting components for the Network and Game subsystems of ARENA, we effectively restricted the alternatives for control flow mechanisms for the game organization part.

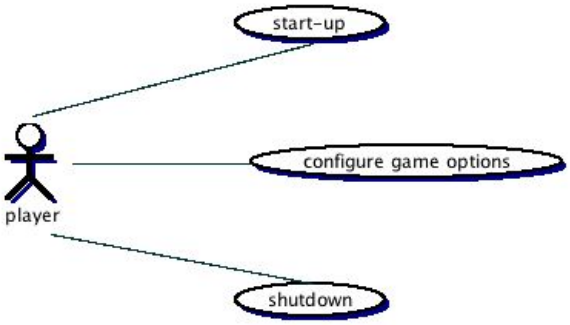

Figure 2.6: Start-up and shutdown

Our system basically use an event-driven flow control. Whenever an event becomes available, it is dispatched to the appropriate object, based on information associated with the event. The GameManager class creates three event dispatcher classes: a remote dispatcher that relays events to all other peers, a local dispatcher that relays events to the GameManager and the ObjectManager.

We use also a threaded control flow in the PeerConnection. Each PeerConnection is handled by a thread. The system can create an arbitrary number of threads . The PeerConnection class is responsible for serializing the outgoing events and sending them over the network connection as well as for deserializing the incoming events from the network connection and sending them to the local event dispatcher.

2.7 Boundary conditions

2.7.1 Prerequisites

For the peer to successfully run and participate in network games, there must be a physical network connection, for example wireless LAN. Further setup steps are not required. The player does not have to enter any network parameters such as IP address and port number.

2.7.2 Start-up and shutdown

Here we describe what has to be done while starting or ending the system.

Start-up and login When a player is logged in to ARENA he has the choice among a variety of network games. If he decides to play SWORD the player can choose to start a new game, to load a game or to join a game (see chapter about Persistent Data Management). In the case the player starts a game he can decide which other players are allowed to join him. After that he has to specify a password for his buddies. The other way round, if a player wants to join a game he has to choose the game and enter the valid password.

Configure game options All game options like the size of the window etc. are configured by the player.

Shut down The game is to shutdown by the player himself. When he leaves the game a window will appear that reminds the player to save either world or adventurer.

2.7.3 Exceptions

Here we describe how our game should react to exceptions that can occur due to problems outside the game itself.

Loss of network connection If a peer loses the connection to the network it notifies the player that the connection has been lost, giving him the chance to restore the network connection by hand. This could for example mean that the player moves to a place where the signal of the wireless LAN is stronger. The player can continue to play while he is not connected to the other peers. As soon as the network is available again the state replication of FRAG merges the players state with the state of the other peers. The player is notified that he is connected again and can continue playing.

When the loss of connection is only short (in the magnitude of a few seconds) the player is not notified at all. The game-play just continues and, in most cases, the player will not notice anything.

Loss of another peer If the network connection to just one other peer is lost and the peer did not quit from the game regularly the player is notified about this fact.

As soon as the peer is available again the player is notified.

As in the case of a total connection loss the player is only notified when the peer is away for more then just a few seconds.

Inconsistent data If the peer notices that his data and the data of another peer is inconsistent it notifies the player. After the player confirms the notification the peer tries to reestablish consistency again.

Corrupted persistent data If the peer notices, that persistent data is corrupted, it stops loading the data and notifies the user.

For example, this could occur when trying to load a saved game from disk, and the file on the disk has been modified since it has been saved.

A Glossary

ARENA
A global software engineering project. The project will develop peer-to-peer multiplayer online games partly on top of the FRAG framework.
Adventurer
The main character in the game that is controlled by the human player.
Analysis
The model of the desired system containing the object model and the dynamic model.
Boundary condition
A special condition the system must handle. Boundary conditions include startup, shutdown and exceptions.
Design goals
Quality criteria that a system should achieve. Design goals are often inferred from nonfunctional requirements and are used to guide design decisions. Examples of design goals include usability, reliability, security and safety.
Examples
illustrate the use of the system with scenarios - instance of a use case
FRAG
A framework that is based on ARENA which especially supports peer-to-peer network communication, distributed object synchronization and message transport for object-based 2D and 3D game worlds
GUI
Graphical user interface. In SWORD the GUI is the interface through which the human player controls the adventurer and gets response messages from the system.
Gladiator
A human player who wants to play a network game based on ARENA is called a gladiator. The application a gladiator starts initializes the PeerDiscovery class and requests a list of running games and peers. It starts the game by calling the GameManager and passing the parameters: IP addresses and player names
Human player
A person who is able to play one or more games. This is the actual human sitting in front of the computer. He or she neither can interact with anything in the game world directly nor can he or she be manipulated directly by software. In order to communicate with the game world a human player acts through the adventurer.
Index
Alphabetically list of all requirement elements for accessing specific elements
Mission
A mission represents a scenario of the game and the steps which the players should accomplish to win the game

A Glossary

ODD
Object Design Document
Problem Statement
Brief introduction to the problem including purpose and scope of the desired system
Peer-to-peer architecture
A generalization of the client-server architecture in which subsystems can act both as clients or servers
Quality Constraints
Nonfunctional requirements
RAD
Requirements Analysis Document
Requirements
Define the problem domain as relevant to the system including following elements: Actors, User Tasks, Domain Constraints, Quality Constraints on User Tasks
SDD
System Design Document - this document
Services
Functionalities that must be provided by the system (= functional requirements)
Specification
Defines the functionality provided by the system including: Use Cases, Services, Global Functional Constraints, Quality Constraints on Use Cases, Quality Constraints on Services
Subsystem
Division of the system into subsystems
Subsystem Services
Functionalities that must be provided by the subsystem
SWORD
SWORD is a fantasy game which is developed on top the FRAG framework
World Generation Algorithm
Algorithm that is responsible to generate the game world