國立交通大學

資訊科學與工程研究所

博 士 論 文

網路桌面遊戲之模式與系統設計

On the Model and System Design for Online Tabletop Games

研 究 生：徐健智

指導教授：吳毅成　教授

中 華 民 國 九 十 五 年 七 月

網路桌面遊戲之模式與系統設計

On the Model and System Design for Online Tabletop Games

研 究 生：徐健智　　Student：Chien-Chin Hsu

指導教授：吳毅成　　Advisor：I-Chen Wu

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所

博 士 論 文

A Dissertation

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

July 2006

HsinChu, Taiwan, Republic of China

中 華 民 國 九 十 五 年 七 月

網路桌面遊戲之模式與系統設計

學生：徐健智 指導教授：吳毅成博士

國立交通大學資訊工程學系研究所博士班

摘　　要

網路桌面遊戲，如西洋棋、圍棋、橋牌、麻將等，在 Internet 上十分受歡迎。許多桌面遊戲擁有相同的特性，如玩家圍坐在桌子旁，使用撲克牌之類的小物體來玩遊戲。由於桌面遊戲普及性及相似性，讓我們想研究如何加速網路桌面遊戲的發展。根據桌面遊戲的相同特性，我們設計兩個模式。第一個模式允許玩家在一個虛擬的大廳中邀請朋友一起玩遊戲。第二個模式則是為桌面遊戲定義了物件、狀態、以及遊戲的動作。根據這兩個模式，我們設計並實作了了一套網路桌面遊戲發展系統。這個系統的功能包括了網路通訊功能，玩家及遊戲伺服器管理，以及專為桌面遊戲提供的圖形介面支援等等。

在實作的過程中，我們遇到了兩個會造成我們的系統服務暫停或中止的問題。第一個問題是某些對網路讀寫資料的動作可能會造成遊戲伺服器暫時停止運作。第二個問題是當緩衝區溢位 (buffer overflow) 這種錯誤發生時，會讓整個遊戲伺服器程式被破壞，無法繼續提供服務。我們深入研究了這兩個問題並在本論文中提出了我們的解決方法。我們的研究成果可以讓遊戲伺服器程式避免被上述那兩個問題干擾，而能夠正確地持續提供服務。

實際上在台灣和香港已經有數個遊戲網站採用我們的系統。近年來，這些遊戲網站已經擁有超過百萬的會員，而最高的同時上線玩遊戲人數，也屢次超過了一萬人，足以說明我們的系統的實用性及穩定性。

E
S
A
1896

On the Model and System Design for Online Tabletop Games

Student: Chien-Chih Hsu Advisor: I-Chen Wu

Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University

Abstract

Online tabletop games, such as Chess, Go, Bridge, Mahjong, etc, are popular in the Internet. Many online tabletop games share the same characteristics, e.g., players play around tables using physical objects such as cards. Due to the popularity and similarity of tabletop games, we are motivated to facilitate the development of online tabletop games. We design two models for online tabletop games according to the shared characteristics. The first model allows players to invite friends to play games together in a virtual room. The second model defines game objects, game states, and playing operations for tabletop games. Based on the two models, we design and implement a system for developing online tabletop games. The functionalities of the game system include network communication, player management, game server management, and graphical support for online tabletop games.

While implementing the game system practically, we encounter two issues on which the services of our system for online tabletop games are blocked or disrupted. The first issue is that some network I/O operations may block servers in our system unexpectedly for a long time. The second issue is that buffer overflow failures may crash servers in our system and therefore disrupt the services. In this dissertation we propose solutions to the two issues respectively so that our game system can support services smoothly and correctly without

suffering form the two issues.

In practice, our game system has been used in some commercial game sites in Taiwan and Hong-Kong. These game sites have attracted more than one million people to register as members and have supported up to 10,000 concurrent players.

誌　　謝

能夠取得博士學位，我要感謝許多人的支持與幫忙。首先感謝我的指導教授吳毅成老師，在我研讀碩士及博士期間，在學業上給我許多指導與教誨，在生活也給我許多關懷與幫助。

其次我要感謝我的論文口試委員朱正忠教授、黃世昆教授、許舜欽教授、郭譽申教授、葉義雄教授、蔡文能教授與顏士淨教授（以上按姓氏筆劃排列），他們對我的論文的指導與建議，對我日後的研究工作有著莫大的助益。

在吳毅成老師門下學習的期間，許多學長姐、同學、及學弟妹曾與我一起合作研究、學習新知，也分享了不少生活上的喜怒哀樂。感謝他們給我許多建議和鼓勵，陪我渡過了這段時光。

最後我要感謝我的父母、妻子、以及其他家人，他們在精神及物質上的支持與鼓勵，使我得以安心進修，完成這個學業。謹將這篇論文，獻給我最摯愛的家人。

Contents

摘　　要 I

ABSTRACT III

誌　　謝 V

CONTENTS VII

LIST OF FIGURES IX

CHAPTER 1　INTRODUCTION 1

1.1　BACKGROUND 1

1.2　MOTIVATION 3

1.3　ORGANIZATION OF DISSERTATION 6

CHAPTER 2　MODELS FOR TABLETOP GAMES 7

2.1　THE GROUPED ONLINE TABLETOP GAME (GOTG) MODEL 7

2.2　THE TABLETOP GAME (TG) MODEL 9

2.2.1 The Simplified Tabletop Game (STG) Model *9*

2.2.2 The General TG Model *17*

CHAPTER 3　THE DESIGN AND IMPLEMENTATION OF THE GOTG SYSTEM 21

3.1　THE BASIC FUNCTIONALITIES OF THE GOTG SYSTEM 21

3.2　THE SPECIFIC SUPPORT FOR TABLETOP GAMES 25

3.3　A GOBANG BASED ON THE TG FRAMEWORK 27

3.4　AN UNIVERSAL TABLETOP GAME SYSTEM 31

CHAPTER 4　RELATED ISSUE 1: RESOLVING PROBLEMS OF BLOCKING I/O OPERATIONS FOR SERVER PROGRAMMING 35

4.1　SERVER PROGRAMMING FOR INTER-USER COMMUNICATION 35

4.2　THE EVENT-DRIVEN PROGRAMMING MODEL 38

4.3　OUTPUT BLOCKING PROBLEM AND SOLUTION 40

4.3.1 Output Blocking Problem *41*

4.3.2 Solution to the Output Blocking Problem *41*

4.4　REQUEST BLOCKING PROBLEM AND SOLUTION 45

4.4.1 Request Blocking Problem *45*

4.4.2 Solutions for HTTP Access Requests *47*

4.4.3 Solutions for Other Service Requests......50
4.5 EXPERIMENTS......55

CHAPTER 5 RELATED ISSUE 2: FAST RECOVERY FROM OVERFLOW FAILURES FOR NON-DISRUPTIVE GAME SERVICES..59

5.1 THE BUFFER OVERFLOW PROBLEM......59
5.2 THE DESIGN AND IMPLEMENTATION OF BODAR......61
5.2.1 Integration of Address Space Permutation and Trapping OOB by Unallocated Address Space......61
5.2.2 Guarding Buffers with Addresses within Unallocated Regions......63
5.2.3 Faulty Address Resolution......65
5.2.4 Source Code Transformation for the Guarded OOB Instrument......66
5.3 RESULTS......68
5.3.1 Stack Buffer Allocation Overhead......68
5.3.2 Security Tolerance and Availability Evaluation......69
5.3.3 Efficiency Evaluation......71
5.4 DISCUSSIONS......75
5.4.1 The Utilizable Virtual Address Space......75
5.4.2 Limitations......77

CHAPTER 6 CONCLUSION......79

6.1 SUMMARY OF CONTRIBUTIONS......79
6.2 FUTURE WORK......83
6.2.1 The Improvements on the GOTG System......83
6.2.2 GAML and History Authoring Systems......84
6.2.3 The Improvement on BODAR......84

REFERENCES......87

List of Figures

Figure 2.1. The Grouped Online Tabletop Game (GOTG) model. 8
Figure 2.2. A game play of Tic-tac-toe. 12
Figure 2.3. (a) The appearance before moving heart four and (b) after moving heart four. 19
Figure 3.1. The architecture of the GOTG system. 21
Figure 3.2. The class diagram of the TG framework. 26
Figure 3.3. A screenshot of the TG framework based Gobang. 28
Figure 3.4. The main flowchart of the TG framework based Gobang. 29
Figure 3.5. Part of the TG framework based Gobang program. 30
Figure 3.6. Initializing a game. 31
Figure 3.7. Defining a new TG game. 32
Figure 3.8. A game appearance for player one. 33
Figure 4.1. Class diagram of the Reactor pattern. 39
Figure 4.2. Class diagram of the event-driven framework with the output buffering mechanism. 42
Figure 4.3. Sequence diagram for accepting a new client. 44
Figure 4.4. Sequence diagram for handling output buffering. 44
Figure 4.5. Class diagram of the event-driven framework with service brokers. 48
Figure 4.6. Sequence diagram for establishing a connection and sending an HTTP request. 49
Figure 4.7. Sequence diagram for handling responses. 50
Figure 4.8. Handling database requests using multi-threads. 51
Figure 4.9. Class diagram of helper processes. 52
Figure 4.10. Collaboration diagram of handling services in a helper process. 52
Figure 4.11. Case of deploying helper processes. 54
Figure 4.12. The deployment of the evaluating the output buffering mechanism. 56
Figure 4.13. The averaged top-100 response time. 56
Figure 4.14. The averaged top-100 response times vs. the blocking times. 57
Figure 5.1. The buffer organization in the BODAR system. 64
Figure 5.2. Converting stack buffers to heap buffers. 67
Figure 5.3. The stack buffer allocation overhead. 68
Figure 5.4. Apache http server 1.3.34 (with mod_mylo 2.1) runs under attacks (part 1). 70
Figure 5.5. Apache http server 1.3.34 (with mod_mylo 2.1) runs under attacks (part 2). 70
Figure 5.6. Thttpd 2.23beta1 runs under attacks (part 1). 71
Figure 5.7. Thttpd 2.23beta1 runs under attacks (part 2). 71
Figure 5.8. Apache http server 1.3.34 and thttpd 2.23beta1 runs without attacks. 72
Figure 5.9. Sparse Degree Measurement. 72
Figure 5.10. Use GnuPG to perform RSA encryption. The source file sizes are 0.5M, 1M,

1.5M, 2M, 2.5M, and 3M bytes files respectively. 73
Figure 5.11. Use GnuPG to perform RSA decryption. The decrypted file sizes are 0.5M, 1M, 1.5M, 2M, 2.5M, and 3M bytes respectively. 73
Figure 5.12.Use access-time.awk to analyze 50K-lines, 100K-lines, 150K-lines, 200K-lines, 250K-lines, and 300K-lines squid access log respectively. 74
Figure 5.13. Use proxy_stats.awk to analyze 50K-line, 100K-lines, 150K-lines, 200K-lines, 250K-lines, and 300K-lines squid access log respectively. 74
Figure 5.14. Use scalar.awk to analyze 50K-lines, 100K-lines, 150K-lines, 200K-lines, 250K-lines, and 300K-lines squid access log respectively. 74
Figure 5.15. Sparse Degree of CPU bound programs. 75
Figure 5.16 The memory layout of a BODAR-enabled process on FreeBSD 5.4 on an AMD-64 machine or an Intel IA-32 machine. 75
Figure 6.1. The screenshot of a game coordinator in the CYC Game League. 82
Figure 6.2. The screenshot of a Chinese Chess game in the CYC Game League. 83

Chapter 1 Introduction

In this chapter, we present a brief introduction to online games. Among these games, we further discuss online tabletop games which are a kind of popular online games in the Internet.

1.1 Background

With the rapid development and the fast-growing user base of the Internet, online games become one of the most important applications in the Internet communities. The market as well as the player number of online games are growing up over the past decade. There are many categories of online games. Some popular categories of online games are briefly introduced as follows.

- Turn-based strategy games: Strategy games are games that the decision-making skills of players significantly affect the game result. A player of a turn-based strategy game is allowed to think for a period of time before committing a game action. The examples of turn-based strategy games include the Civilization Net series [25] and the Total War series [69].
- Real-time strategy games: Real-time strategy games refer to a certain computer strategy games that the action in the games is continuous and the players have to make decisions while the states of games constantly progress. The examples of real-time strategy games include the AOE series [42] and the Warcraft series [9].
- First-person shooter games: First-person shooter games are a type of fighting video or computer games. In such a game, the screen simulates the first-person view. The examples of first-person shooter games include the Doom series [34] and the Half Life series [84].
- Role-playing games: A role-playing game is a kind of game in which a player pretends the

role of a character and collaborates with others to create narratives. The game progresses according to a predefined system of guidelines and rules. The examples of role-playing games include the Diablo series [10] and the Baldur's Gate series [8].

- Tabletop games: Tabletop games is a general term that refers to card games (e.g., Bridge, Big2, and Showhand), board games (e.g., Chess, Chinese chess, and Othello), tile-based games (e.g., Mahjong) and other games that are normally played on a table with some small objects (such as cards or pieces).

Although tabletop games are relatively simple among these game categories, they are still popular in the Internet. For example, Yahoo! Games [95] that supports many tabletop games is a very popular game site. Millions of people register as the its members and more than 40000 of the members log on it concurrently to play tabletop games anytime in 2006. Besides, a survey undertaken in 2005 by the Market Intelligence Center of the Institute for Information Industry of Taiwan reports that the market share of online tabletop games in Taiwan is about several billions in 2005 and 2006 [35].

Many tabletop games have similar characteristics. For example, players usually play around tables using physical objects such as cards. We summarize the shared characteristics of tabletop game as following.

- Players of a game sit around a table.
- They play the game by manipulating physical objects on the table. For example, players who play Chess move Chess pieces on a chessboard on the table.
- They may hold objects in their private areas to keep these objects secret. For example, players that play Bridge may hold cards in their own hands.
- All objects are finite and never disappear on the table.

The summarized characteristics of tabletop games focus on the objects used in tabletop

games and operations that manipulate these objects. We believe that a model for tabletop games can be designed according to these shared characteristics and a reusable software architecture for tabletop game development can be constructed based on the model.

Since the model mentioned above focuses on objects and operations in tabletop games, it does not consider the matchmaking functionality, that is, searching partners to play games together. However, the matchmaking functionality is very important for online games since players would like to play games in an online game system if they can find their partners in the game system easily. In order to support the functionality, we can design another model that depicts the following scenario:

- There are many tables in a large room.
- After a player P enters the room, P can search or wait for P's friends.
- If P finds that some of P's friends have sat around a table, P may sit around the table and play games with them.
- P can sit around an empty table and wait for P's friends to sit down and then play games together.

Based on the model, the matchmaking functionality for an online tabletop game system can be implemented in an easy way. In fact, some popular online tabletop game sites Yahoo! Games [95] and Acer Game Zone [30] have used similar model to implement their matchmaking functionality.

1.2 Motivation

Due to the popularity and similarity of online tabletop games, we are motivated to facilitate the development of tabletop games. We expect to design and implement an online tabletop game system practically based on the similarity mentioned in Section 1.1 so that an

online tabletop game can be easily developed. We first survey the past work for tabletop games below.

There are several popular online tabletop game sites in the Internet. Some of these sites are dedicated to single tabletop game. For example, Free Internet Chess Server (FICS) [24] is dedicated to Chess while Internet Go Server (IGS) [53] is dedicated to Go. Other tabletop game sites, such as Yahoo! Games [95] and Acer Game Zone [30], support a series of tabletop games. Although these game sites are well-designed and attractive to players, they rarely describe the design of their systems in detail.

A research topic related to tabletop games is to develop artificial intelligence systems that can play games with human. For example, the IBM Deep Blue project [33] developed the epochal Chess game system that won the World Chess Champion in 1997. Besides, the research on playing Chinese Chess [91, 96], Go [77], and Connect6 [92, 93] also made achievements. Since the research topic focuses on artificial intelligence, they seldom discuss about how to develop tabletop games.

The research in [55, 56] presented a system named Extensible Graphical Game Generator (EGGG) that can use to develop tabletop games. EGGG defines a high-level script language that allows users to design games. The rules, graphical content, and network functionality of an online tabletop game can be designed with the script language. The authors of EGGG expect that users can use the script language to design a game easily, therefore they claim that a game is easy to be designed and debugged. However, they did not formalize their game model so that it is unknown about how general their systems are. Besides, EGGG does not provide the matchmaking functionality so that the users must develop the functionality if needed.

Eeik et al. proposed a multiplayer board game framework [49] for developing online

board games. It supported a set of reusable components for network communication and graphic user interface. Besides, it also provides default control flow for board games. Although the framework provides some supports board games, it does not support other types of tabletop games, such card games or tile-based games.

In the dissertation, we try to facilitate the development of online tabletop games in the following way. First we design two models for online tabletop games. The first one named Grouped Online Tabletop Game (GOTG) model that allows players to search for their friends in a virtual room and then play games with them in the room. The second one named the Tabletop Game (TG) model that defines game objects, game states, and playing operations for tabletop games. For the TG model, we formalize the definition of tabletop games and related terminologies and then prove that it is general for all tabletop games.

Based on the TG and GOTG models, we implement a system also named GOTG for developing online tabletop games. The GOTG system provides services over the Internet that allow players to gather in groups to play tabletop games. Besides, it also supports the service for matchmaking so that players can search for their friends to play games together. For developing a tabletop game, the GOTG system provides a framework based on the TG model to help developers handling objects used in the game and operations that manipulate these objects.

While implementing the GOTG system practically, we encounter two issues that may block or disrupt the services of our system. The two issues are briefly described as follows.

- Resolving problems of blocking I/O operations for server programming

  The game servers in the GOTG system use the event-driven model for the concurrent message (or event) handling. However, using blocking I/O operations in an event-driven server may freeze the whole server so that the services of the server stop for an

unexpected period of time. We have identified two common blocking problems due to the use of blocking I/O operations and provide solution to them. The solution have been integrated into the GOTG system so that the game servers in the GOTG system can be immune from the two blocking problems.

- Fast recovery from overflow failures for non-disruptive game services

  The GOTG system provide services that allow tabletop games to be played over the Internet. These services rely on the collaboration of several kinds of servers, including game servers, web servers, database severs, etc. However, the services may disrupt if some of these servers crash. To continuously provide the services, the servers should try to recover from failures timely. In the dissertation, we address the buffer overflow failure that may disrupt some of these servers. To prevent buffer overflow failures from disrupting servers, we develop a technique to detect and recover from buffer overflow failures efficiently.

Since we expect the GOTG system to continuously provide services for players, we try to solve the two issues. We present our solutions in the dissertation that make our system provide services correctly and smoothly without suffering from the two issues.

1.3 Organization of Dissertation

Chapter 2 presents the models for online tabletop games. Chapter 3 introduces the design and implementation of the GOTG system. Chapter 4 describes the influence of blocking I/O operations on server programming and the solution to this issue. Chapter 5 describes the influence of buffer overflow failures on game services and the solution to this issue. Finally, Chapter 6 presents our conclusions.

Chapter 2　Models for Tabletop Games

In this chapter, we present two models for tabletop games. The first one allows players to search for their friends and then gather in groups to play games together. The second one defines objects used in tabletop games and operations that manipulate these objects. The two models can help us to design and implement a tabletop development system in practice.

2.1　The Grouped Online Tabletop Game (GOTG) Model

For most tabletop games, a game session on a table is independent from game sessions on other tables. That is, players' actions on a table do not affect any games on other tables. Therefore, in order to allow tabletop games to be played over the Internet, it is possible to run many isolated game servers and each of them simulates a table. In such configuration, a player can connect to one of these servers and then play games with other players on the same server.

However, the configuration is not convenient for most players because they must know a list of game servers in advance. Even a player P know the list, P does not know if some of P's friends have logged on a server before P logs on it. Therefore, P may spend much time to search for friends.

In order to avoid such problem, we present a model named the Grouped Online Tabletop Game (GOTG) for online tabletop games. Figure 2.1 shows the GOTG model in which tables connect to a game coordinator and players connect to a table or the game coordinator. In fact, the model depicts the environment under which there are many tables and players in a large room where the players may search for friends or sit around one of the tables to play games. The behavior of players in the model is summarized as follows.

- A player P must log on the game coordinator first.

- After logging, P can browse the list of tables and players to find P's friends. The list is stored and maintained by the game coordinator.
- P may join some table.
- When the table is full of players, a game starts.
- Or P can sit around an empty table and wait for P's friends to sit down and play.
- P may leave the current table and then join another table.

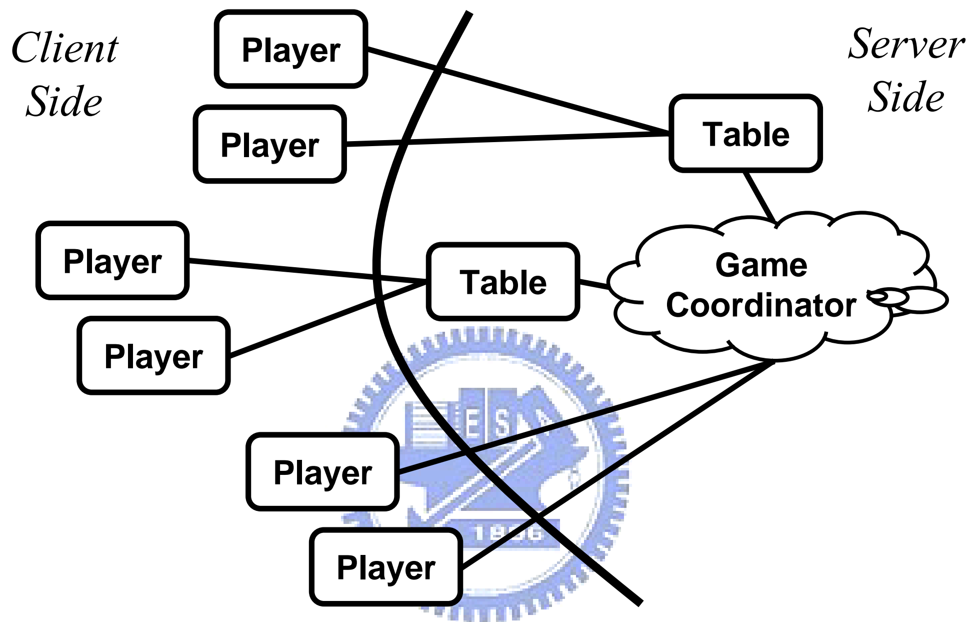


Figure 2.1. The Grouped Online Tabletop Game (GOTG) model.

The GOTG model is based on the client-server model in which players are in the client side and tables and the game coordinator are in the server side. We adopt the client-server model rather than the peer-to-peer model for the following two reasons.

1. Store critical data in server side to provide further services. For example, players' game scores are generally stored in server side. If a game company keeps these game scores, it can hold a game competition and invite players with top scores to join the competition in order to popularize its games.

2. Process important data in server side to keep it secure. For example, the game scores should be stored and computed in server side. If they are stored and computed in client

side, they may be illegally modified and the fairness of the games would be suspected.

Conceptually there is one and only one game coordinator in the GOTG model. Practically the game coordinator may be distributed over different servers to increase scalability.

2.2 The Tabletop Game (TG) Model

This section formally defines the Tabletop Game (TG) model. For convenience those tabletop games that follow the TG model are called TG games. First, for simplicity, Subsection 2.2.1 defines the simplified TG (STG) model for tabletop games in which all objects are publicly put on the table. In addition, it is proved that some game systems can be devised to play all Simplified TG (STG) games. Second, Subsection 2.2.2 defines the general TG model that allows players to hold game objects in their own private areas. Similarly, it is shown that some game systems can be devised to play all the TG games.

2.2.1 The Simplified Tabletop Game (STG) Model

A STG game is $G = (A, O, S, M)$ as defined as follows.

- A is a set of two-dimensional coordinates or a two-dimensional area representing the game table.
- O is a finite set of n *game objects*. Namely, $O = \{o_1, o_2, o_3, \ldots, o_n\}$, where each object o_i has m_i faces, $\Phi_i = \{\phi_{i1}, \phi_{i2}, \phi_{i3}, \ldots, \phi_{i,m_i}\}$. Here, i is the *object index*. Now, with A and O, all the possible table states $\Gamma(A, O)$ can be derived as follows: $\Gamma(A, O) = O_1 \times O_2 \times \ldots \times O_n$, where $O_i = A \times Z \times \Phi_i$ and Z is the set of Z-order values (for simplicity, let them be real numbers, R). O_i is said to be the set of *states of object* o_i. For each table state $q = ((p_1, z_1, f_1), (p_2, z_2, f_2), \ldots, (p_n, z_n, f_n))$ where $1 \le f_i \le m_i$ and for each object o_i, the

object state (p_i, z_i, f_i) indicates that this object o_i switches to the face ϕ_{i,f_i} at position p_i with Z-order z_i (representing the distance to viewers; note that for simplicity of discussion the larger the Z-order the closer the object appears). One key point of STG games is that no objects disappear on the table once games have been set up.

- S is a set of legal *game states*. Namely, $S = \{(start), (end)\} \cup S^{-}$, where S^{-} is a subset of $\Gamma(A,O)$. For each game, players start from the state (*start*) and end at the state (*end*).
- M is a set of legal *game operations*, representing the game rule. Namely, M is a subset of $S \times S$, where each legal operation (s, s') indicates that this operation changes the game state from s to another s'. Those operations with $s = (start)$ are called *initial* operations; and those with $s' = (end)$ are *final* operations.

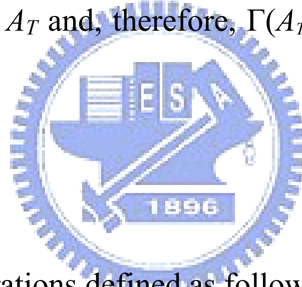
In the above game G, a sequence of game states $(q_1, q_2, q_3, \ldots, q_g)$ is called a *game play interval*, if each neighboring pair of states (q_i, q_{i+1}) in the sequence is a legal game operation in M. A sequence of game states $(q_1, q_2, q_3, \ldots, q_g)$ is called a *game play* in G, if q_1 is (*start*), q_g is (*end*), and the sequence is a game play interval.

For each game state, its *game appearance* is the image that the players can see on the table. As for the two special game states, (*start*) and (*end*), their appearances are defined to be different from any other game states. For each of all the other game states (i.e., table states) $q = ((p_1, z_1, f_1), (p_2, z_2, f_2), \ldots, (p_n, z_n, f_n))$, the appearance is the image canvas painted in the following steps:

1. Sort the objects by using their Z-order as the major keys and their object indices as the minor keys in the ascending order.
2. According to the sequence (sorted in Step 1), repeatedly paint each object o_i with face ϕ_{i,f_i} at position p_i on the image canvas.

Let us illustrate the above definition by the game Tic-tac-toe $G_T = (A_T, O_T, S_T, M_T)$ as follows.

- A_T is a table with 3 × 3 grids (at positions $d_{11}, d_{12}, d_{13}, d_{21}, \ldots, d_{33}$) plus a place (at position d_{00}).
- O_T has 5 white pieces and 4 black pieces, where each piece has only one face. Each piece is put at one of the 10 positions mentioned above. In this game, we assume white to play first. As above, the set of all the table states $\Gamma(A_T, O_T)$ is $O_1 \times O_2 \times \ldots \times O_9$, where $O_i = A_T \times Z \times \Phi_i$. For simplicity, let the objects with odd object indices be white pieces and the objects with even object indices black pieces. In addition, let all objects have the same Z-order, say 0. For each object, since its Z-order has only one value and it has only one face, O_i can be simplified as A_T and, therefore, $\Gamma(A_T, O_T) = A_T \times A_T \times \ldots \times A_T = A_T{}^9 = \{d_{11}, d_{12}, d_{13}, d_{21}, \ldots, d_{33}\}^9$.



- $S_T = \{(start), (end)\} \cup A_T{}^9$.
- M_T is a set of legal game operations defined as follows:
  - The initial operations: $\{((start), q) \mid q \in A_T{}^9$, all objects of q must be at $d_{00}\}$.
  - The move operations: $\{(q_i, q_j) \mid q_i, q_j \in A_T{}^9$, q_i is the same as q_j except for that one white (black) piece in q_i is moved from d_{00} to a grid which is empty in q_i, if q_i is in the white (black) turn$\}$.

    q_i is in the white turn if the number of white pieces at d_{00} is higher than that of black pieces at d_{00}; otherwise, in the black turn.
  - The final operations: $\{(q, (end)) \mid q \in A_T{}^9$, each grid except for d_{00} has one and only one object$\}$.

    Note that most Tic-tac-toe games end when one line of three pieces has the same

color, but for simplicity, we define that the game ends when all pieces are moved.

A game play of G_T is illustrated in Figure 2.2.

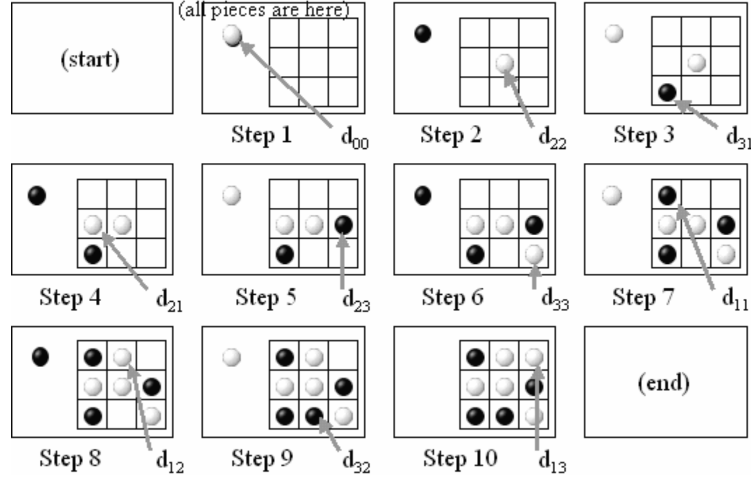


Figure 2.2. A game play of Tic-tac-toe.

For a STG game G, a sequence of game states $(q_1, q_2, q_3, \ldots, q_g)$ is called a *partial game play* of the game play $(q'_1, q'_2, q'_3, \ldots, q'_{g'})$, if $q_i = q'_{m_i}$ for each i where $1 \leq i \leq g$ and $1 = m_1 < m_2 < \ldots < m_g = g'$. Consider two STG games G and G'. If we want a game play ρ in G to be simulated by another ρ' in G', we only need to let the appearances of the game play ρ *look* the same as those of some partial game play of ρ'. More specifically, a game play $(q_1, q_2, q_3, \ldots, q_g)$ in G is *simulated* by another game play $(q'_1, q'_2, q'_3, \ldots, q'_{g'})$ in G', if q_i has the same appearance as q'_{m_i} for each i where $1 \leq i \leq g$ and $1 = m_1 < m_2 < \ldots < m_g = g'$. The operations from q'_{m_i} to $q'_{m_{i+1}}$ are said to simulate the operation (q_i, q_{i+1}).

A STG game G is said to be *simulated* by another STG game G', if for each game play ρ in G there exists some game play in G' that simulates the game play ρ.

A *STG game system* is defined to be a set of STG games. A STG game system is said to be *general*, if all the STG games can be simulated by games in this STG game system.

Now, consider a STG game system $\Psi_C = \{(R^2, O, S_C(O), M_C(O)) \mid \text{for all object set } O\}$,

as described as follows.

- R^2 is the complete two-dimensional space. Note that all game areas A belong to R^2.
- $S_C(O) = \{(start), (end)\} \cup \Gamma(R^2,O)$.
- $M_C(O)$ is the union of the following three sets:
  - – The set of initial operations: $\{((start), q) \mid q \in \Gamma(R^2,O)\}$.
  - – The set of change-object operations: $\{(q, q') \mid q, q' \in \Gamma(R^2,O)$; and the game state q is the same as q' except for that only one object may have different object states in both q and $q'\}$.
  - – The set of final operations: $\{(q, (end)) \mid q \in \Gamma(R^2,O)\}$.

Theorem 2.1 shows that the above game system Ψ_C is general.

Theorem 2.1 *As defined above, the game system* Ψ_C *is general for all STG games.*

Proof. It suffices to prove that for each given STG game $G = (A, O, S, M)$ the STG game $G' = (R^2, O, S_C(O), M_C(O))$ in Ψ_C simulates G. Namely, we want to prove that for each game play ρ in G there exists a game play ρ' in G' simulating the game play ρ.

Let $\rho = ((start), q_1, q_2, q_3, \ldots, q_g, (end))$ in G. First, each object state in game G is also a valid object state in G', since $A \subset R^2$ and $S \subset S_C(O)$ apparently.

Second, for each operation (q_i, q_{i+1}) in ρ, if at least two objects' states are changed, we can insert some game states (in G') such that each operation only changes one object's state. For example, if one operation in G moves both objects o_i and o_j, we insert one more game state in-between to break the operation into two such that the first operation moves object o_i and the second moves object o_j. Obviously, after these game states are inserted, each pair

becomes change-object operations in G'. (Note that all the initial and final operations in G are also legal operations in G'.) Thus, the new sequence of game states including the inserted game states form a game play in G'. This shows that there exists a game play in G' simulating the game play ρ in G. ■

Lemma 2.1 *As described above, if a game system* Ψ *is general for all STG games and another game system* Ψ' *simulates* Ψ, *then the game system* Ψ' *is also general for all STG games.*

Proof. Trivial. ■

From Lemma 2.1 and Theorem 2.1, we can easily prove the generality of another game system Ψ' by letting Ψ' simulate Ψ.

Consider a game system Ψ_{MCC} that is the same as Ψ_C except for that only the following three kinds of change-object operations are legal:

- Move-object operation: A move-object operation is a change-object operation, with changing one object's position only.
- Change-Z operation: A change-Z operation is a change-object operation, with changing one object's Z-order only.
- Change-face operation: A change-face operation is a change-object operation, with changing one object's face only.

Let the game system be $\Psi_{MCC} = \{(R^2, O, S_C(O), M_{MCC}(O)) \mid \text{for all object set } O\}$ and $M_{MCC}(O) = \{(p, q) \mid (p, q) \in M_C(O)$ and (p, q) is an initial, final, move-object, change-Z, or change-face operation$\}$. Theorem 2.2 proves that the game system Ψ_{MCC} is still general for all the STG games.

Theorem 2.2 *The above game system* Ψ_{MCC} *is general for all STG games.*

Proof. From Lemma 2.1 and Theorem 2.1, it suffices to prove that the game system Ψ_{MCC} simulates the general game system Ψ_C, or that for each $G = (R^2, O, S_C(O), M_C(O))$ in Ψ_C the STG game $G' = (R^2, O, S_C(O), M_{MCC}(O))$ in Ψ_{MCC} simulates G. Thus, we will prove that for each game play $\rho = ((start), q_1, q_2, q_3, \ldots, q_g, (end))$ in G there exists a game play in G' simulating the game play ρ.

For each change-object operation $(q_i, q_{i+1}) \in M_C(O)$, where $1 \leq i \leq g-1$, since it only changes an object's three factors, location, Z-order and face, we can break the operation into at most three operations — move-object, change-Z, and change-face operations — in $M_{MCC}(O)$ that simulate the change-object operation. Besides, the initial and final operations are obviously legal in G'. The new sequence of game states including the inserted game states are also a game play in G'. This implies that there exists a game play in G' simulating the game play ρ. ■

Again, consider another new general game system by slightly modifying Ψ_{MCC} as follows. Change-Z and change-face operations are changed to max-Z and next-face operations, respectively.

- Max-Z operation: A max-Z operation is a change-Z operation, but the Z-order of the object is changed to a value larger than any other objects' Z-order.
- Next-face operation: A next-face operation is a change-face operation, but the face of the object o_i is changed from ϕ_{ij} to the next face $\phi_{i,((i+1)\%m_i)}$, where the set of faces of object o_i is $\{\phi_{i1}, \phi_{i2}, \phi_{i3}, \ldots, \phi_{i,m_i}\}$.

Let the game system be Ψ_{MMN} = {(R^2, O, $S_C(O)$, $M_{MMN}(O)$) | for all object set O} and $M_{MMN}(O)$ = {(p, q) | (p, q) ∈ $M_{MCC}(O)$ and (p, q) is an initial, final, move-object, max-Z, or next-face operation}. Then, the game system Ψ_{MMN} is also general for all STG games as shown in Theorem 2.3.

Theorem 2.3 *The above game system* Ψ_{MMN} *is general for all STG games.*

Proof. From Lemma 2.1 and Theorem 2.2, it suffices to prove that the game system Ψ_{MMN} simulates the general game system Ψ_{MCC}, or that for each G = (R^2, O, $S_C(O)$, $M_{MCC}(O)$) in Ψ_{MCC} the STG game G' = (R^2, O, $S_C(O)$, $M_{MMN}(O)$) in Ψ_{MMN} simulates G. Thus, we will prove that for each game play ρ = ((*start*), q_1, q_2, q_3, . . ., q_g, (*end*)) in G there exists a game play in G' simulating the game play ρ.

Let the object set O have n objects and their Z-order be {z_1, z_2, . . . , z_n}. For each change-Z operation ω in ρ, assume that ω changes the Z-order of object o_i from z_i to z'_i. Then, sort these Z-order from {z_1, z_2, . . . , z_{i-1}, z'_i , z_{i+1}, . . . , z_n} to {z^*_1, z^*_2, . . . , z^*_n} in the ascending order. Let o^*_i be the object with Z-order z^*_i. Then, perform the max-Z operation on each object o^*_i once from o^*_1 to o^*_n. Thus, after the n max-Z operations, the appearance is still the same as that for the change-Z operation ω due to the same relative Z ordering. Therefore, the n max-Z operations simulate ω.

For each change-face operation in ρ, assume that it changes a face ϕ_{ij} of object o_i to another face ϕ_{ik}. For the change-face operation, we only need to do the next-face operations on the object ($k - j$) times, if $k > j$, or ($k + m_i - j$) times, if $k < j$ and m_i is the face number of the object o_i. Thus, these next-face operations simulate the change-face operation.

The above implies that there exists a game play in G' simulating the game play ρ. ■

Since the game system Ψ_{MMN} is general for all STG games, it suffices to support the three operations only: move-object, max-Z, and next-face operations. We are more interested in game system Ψ_{MMN} than Ψ_C or Ψ_{MCC}, because it is much easier to implement move-object, max-Z, and next-face operations than other operations, as shown in Section 3.4 (below).

2.2.2 The General TG Model

In the STG model, we assume that all players can see all the game objects on the table. However, for the games such as Bridge, players can only see their own cards that others cannot see. Therefore, in the model of TG games, we need to extend one area to multiple areas, the one for all players, called the *public area*, and every other for one player or a certain number of players, called the *private area*.

A p-private-area TG (p-TG) game is $(A^{(p+1)}, O, S, M)$, as defined as follows.

- $A^{(p+1)}$ is a set of 2-D areas, $\{A_0, A_1, A_2, . . . , A_p\}$. A_0 is the public area and A_i is the i-th private area.

- O is the same as the definition of the object set for STG games.
  Similarly, we define the set of table states $\Gamma(A^{(p+1)}, O) = O_1 \times O_2 \times . . . \times O_n$, each $O_i = A^{(p+1)} \times Z \times \Phi_i$. For $A^{(p+1)}$, we use (b, p) to indicate the position p in area A_b. Thus, we use (b_i, p_i, z_i, f_i) to indicate the state of object o_i, instead.

- S is $\{(start), (end)\} \cup S^-$, where S^- is a subset of $\Gamma(A^{(p+1)},O)$.

- M is similar to the definition for STG games.

In each game state $q = ((b_1, p_1, z_1, f_1), (b_2, p_2, z_2, f_2), . . . , (b_n, p_n, z_n, f_n))$, the i-th object is

put at position p_i in area A_{b_i} with face ϕ_{i,f_i} and Z-order z_i.

A *TG game* is a *p*-TG game, if there are *p* private areas in the game. Obviously, STG games can be viewed as 0-TG games.

For each game state, its game appearance is the images that players can see. For example, player *i* can only see the public area A_0 and its private area A_i. Thus, a game's appearance should include the appearances of the $p + 1$ areas. More specifically, for a game state $q = ((b_1, p_1, z_1, f_1), (b_2, p_2, z_2, f_2), \ldots, (b_n, p_n, z_n, f_n))$, the appearance of *q* has $p + 1$ image canvases and it is painted in the following steps:

1. Sort the objects in the same way as Step 1 in Subsection 2.2.1.
2. According to the sequence (sorted in Step 1), paint each object say o_i with face ϕ_{i,f_i} at position p_i on the b_i-th image canvas.

An example of a poker game is illustrated in Figure 2.3. Figure 2.3(a) and 2.3(b) show the appearances before and after moving heart four from the common area A_0 to the private area A_2.

Similarly, we define that a game play in a *p*-TG game is *simulated* by another *p*-TG game, in the same way as STG games. A *p*-TG game *G* is *simulated* by another *p*-TG game *G*', if for each game play ρ in *G* there exists some game play in *G*' that simulates the game play ρ. A *p*-TG game system is defined to be a set of *p*-TG games. A *p*-TG game system is *general*, if each *p*-TG game is simulated by some game in this *p*-TG game system.

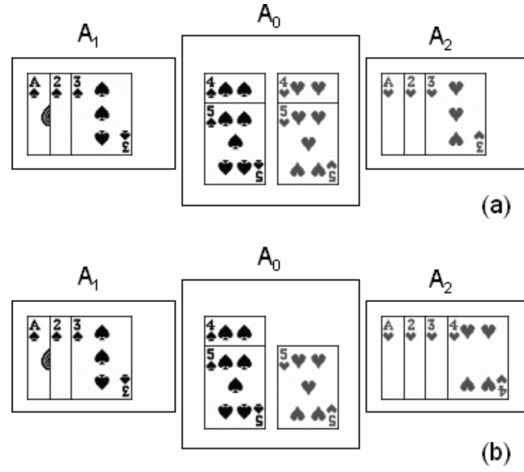


Figure 2.3. (a) The appearance before moving heart four and (b) after moving heart four.

Now, consider a p-TG game system $\Psi_C^{(p)} = \{((R^2)^{(p+1)}, O, S_C^{(p)}(O), M_C^{(p)}(O)) \mid$ for all object set $O\}$, as described as follows.

- $(R^2)^{(p+1)}$ has p + 1 areas, each of which is the whole two-dimensional space, R^2.
- $S_C^{(p)}(O) = \{(start), (end)\} \cup \Gamma((R^2)^{(p+1)}, O)$.
- $M_C^{(p)}(O)$ is the union of the following three sets:
  – The set of initial operations: $\{((start), q) \mid q \in \Gamma((R^2)^{(p+1)}, O)\}$.
  – The set of change-object operations: $\{(q, q') \mid q, q' \in \Gamma((R^2)^{(p+1)}, O)$; and the game state q is the same as q' except for that only one object may have different object states$\}$.
  – The set of final operations: $\{(q, (end)) \mid q \in \Gamma((R^2)^{(p+1)}, O)\}$.

Similar to Theorem 2.1, Corollary 2.1 shows that the above game system $\Psi_C^{(p)}$ is general for all p-TG games.

Corollary 2.1 *The above game system* $\Psi_C^{(p)}$ *is general for all p-TG games.*

By using the techniques similar to those used in Subsection 2.2.1, we can break a change-object operation into several move-object, max-Z, and next-face operations that simulate the change-object operation. Now, let $\Psi_{MMN}^{(p)}$ be the same as $\Psi_{C}^{(p)}$ except for that the set of legal operations are all of these three operations. Thus, we can obtain Corollary 2.2.

Corollary 2.2 *The above game system* $\Psi_{MMN}^{(p)}$ *is general for all p-TG games.*

Consider a TG game system $\Psi_{C}^{*} = \bigcup_{i=0}^{\infty} \Psi_{C}^{(i)}$. The TG game system Ψ_{C}^{*} is general for all TG games as in Theorem 2.4 (below). Similarly, the TG game system $\Psi_{MMN}^{*} = \bigcup_{i=0}^{\infty} \Psi_{MMN}^{(i)}$ is also general for all TG games as in Theorem 2.4 (below).

Theorem 2.4 *Both TG game systems* Ψ_{C}^{*} *and* Ψ_{MMN}^{*} *(described above) are general for all TG games.*

Proof. Trivial. ■

Chapter 3 The Design and Implementation of the GOTG System

In this chapter we discuss the design and implementation of the GOTG system. First, we present the basic functionalities of the GOTG system. Then, we present a framework based on the TG model for developing the Graphic User Interface (GUI) of tabletop games. Next, we demonstrate how to develop a Gobang based on the framework and the GOTG system. Finally, we present an universal tabletop game system that allows users to design and play tabletop games.

3.1 The Basic Functionalities of the GOTG System

The GOTG system is designed based on the TG and GOTG models for developing online tabletop games. Figure 3.1 (below) shows the architecture of the GOTG system. According to the functionalities, the system can be divided into several modules. The shadowed modules are the game specific part that game developers need to implement. Note that the network module and player management module are implemented in both client and server sides since their functionalities need the collaboration of both sides.

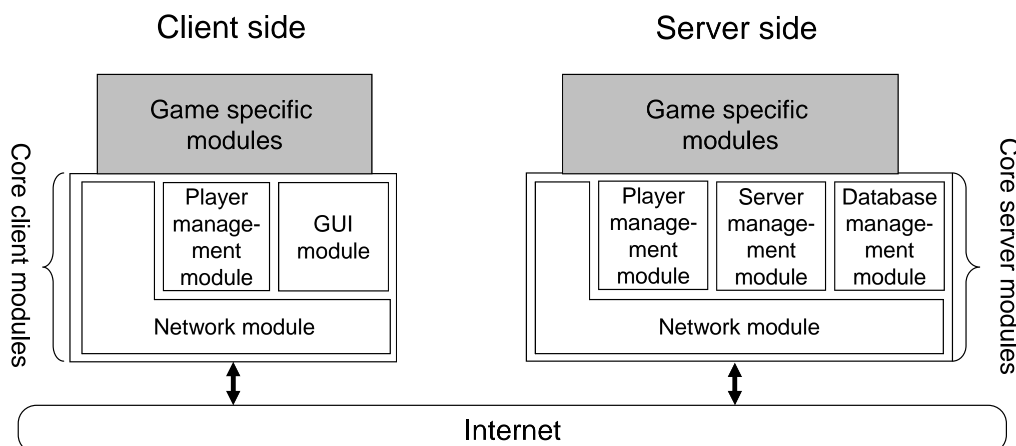


Figure 3.1. The architecture of the GOTG system.

The functionality of each module is introduced below.

- Network module

  The communication ability over the Internet is essential for all online games. The network module provides the communication ability for GOTG-based games. Other modules in the GOTG system (including game specific modules) send and receive data through this module.

  A simple application layer protocol is defined and realized by the network module based on the TCP/IP [18, 19, 73]. The protocol specifies a message format so that other GOTG modules and game-specific modules can send and receive messages through the network module. If an incomplete message is received, the network module stores it until the remainder is received. Once a message is completely received, the network module dispatches it to the destination module.

  The network module can encrypt and decrypt the messages in order to prevent from eavesdropping. For the performance consideration, this module provide a symmetric key algorithm based on the exclusive or (abbr. XOR) operation. That is, the encryption performs the XOR operation on the plain text (original data) and a key while the decryption performs the XOR operation on the cipher text (encrypted data) with the same key. The detail process is described in [39].

  To use a symmetric key algorithm, the peers that exchange cipher text must agree on a key in advance. Currently, the GOTG system uses the password of a player as the key in the XOR-based cryptography. The password of the player is transmitted to a game server when the player logins. The data exchanged in the login process is transferred with the https protocol [61] to keep it secret.

- Player management module

  This module is design for the membership management and matchmaking functionalities. For the former functionality, this module control provides interfaces that allow a user to

register as a member. Besides, this module also provides interfaces that allow game developers to update the newest state of a member. To support the membership management functionality, this module controls the database management module (described below) to store the states of members.

For the matchmaking functionality, this module maintains a list of players that login the GOTG game system. To maintain such a list, this module manages the process of player login and logout. With the help of the list, it is easy for players to find their friends or opponents. Besides, this module also helps a player to join or create a game before playing and leave a game after playing.

- Graphic User Interface (GUI) module

  This module includes two parts. The first part is the interfaces that allow the player management module to refresh its recent status, i.e. the list of players that currently login the GOTG game system. A player can control the player management module through the interfaces in order to join or create a game. Several implementations of the interfaces are also included so that the developers can use them directly.

  The second part is a framework that is designed based on the TG model for tabletop games. The framework is introduced in Section 3.2.

- Database management module

  A game company generally store the players' records and other states in the database to provide further services or for business. Therefore, the we designe this module to help developers accessing SQL-based database. To hide the detail of SQL, this module provide a simple interface to access database so that the developers do not need to handle SQL in many cases. Besides, this module maintains a set of connections to database servers in order to reduce the overhead of connection establishment. Finally, this module can be configured to log update queries for auditing.

- Server management module

  This module provides an interface for the managers of game companies to monitor and control game servers in the GOTG system. The managers can use the interface to query information of the game servers, such as the player number and the usage of some resources. Besides, the managers can use the interface to control the game server to do the following matters: broadcasting messages to all or specific players, kicking out a player who does something bad, etc.

  Currently, the managers communicate with the server management module through a TCP socket. That is, this module creates a TCP socket that allows the managers to login and then monitor or control the game server remotely.

The server side of the GOTG system is implemented on the Linux/FreeBSD platforms with the C/C++ languages. The main reason why choose Linux/FreeBSD is the consideration of cost. Linux/FreeBSD are free and stable enough for many applications. Besides, there are also many powerful and open source tools on Linux/FreeBSD for system administration, network administration, programming, etc.

The main reason why choose the C/C++ languages for the server side programs is the consideration of performance. In order to reduce the cost, it is feasible to serve as many players on a game server as possible. Therefore, the author choose the C/C++ languages to maximize the performance of game server programs so that the number of game servers could be reduced.

The client side of the GOTG system is developed on the Java platform due to the availability of Java. Java is available on many operating systems (such as Windows, Linux, and Mac OS) and different hardware (such as PC, PDA, and mobile phone). The wide-spread of Java make the ideal of “write once, run everywhere” easier to realize. The author designs the Java-based client side programs of the GOTG system for the portability to as many

operating systems and hardware as possible.

3.2 The Specific Support for Tabletop Games

In this section we develop a framework named the Tabletop Game (TG) framework according to the TG model. The goal of this framework is to facilitate the development of the graphic user interface (GUI) for tabletop games. Practically, the TG framework is developed in Java and implemented as a part of the GOTG system so that the TG framework based games can also benefit from the advantages of the GOTG system.

Since the TG framework is dedicated for the GUI of tabletop games, the developers need to program other parts of the games, such as game state handling and rule checking. In order to simplify the description of TG framework, we assume that each TG framework based game has an object named `Game_Status` that is responsible for handling the states and controlling other parts of the game, including the TG framework. That is, the `Game_Status` object controls the TG framework to draw the appearance of the game. Besides, the TG framework reports a player's operations to `Game_Status` so that it can refresh the game states and decide the end of the game.

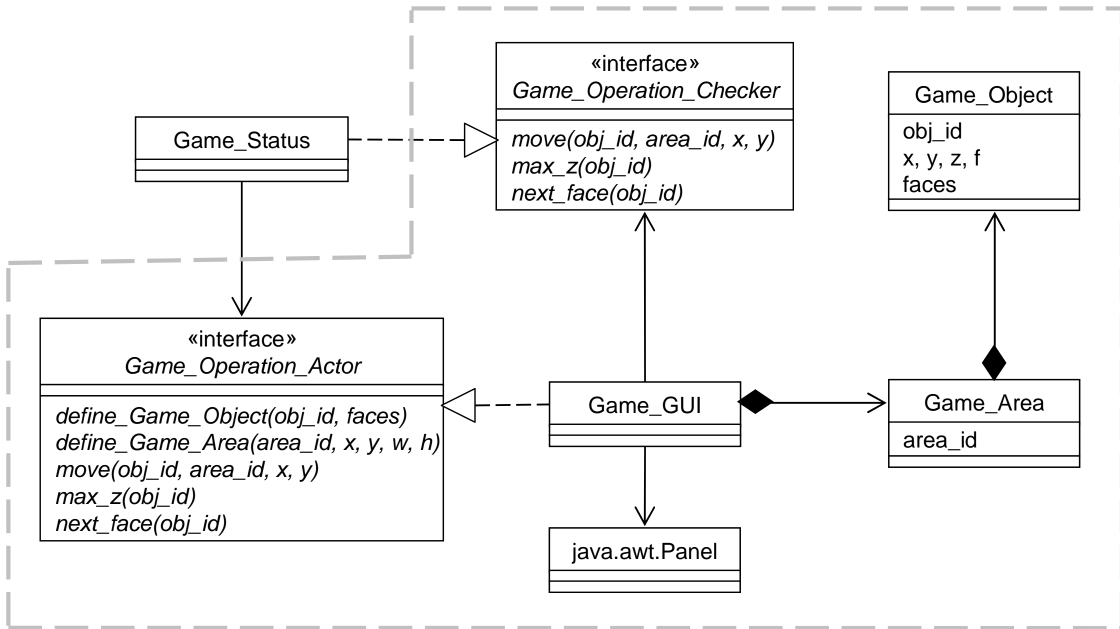


Figure 3.2. The class diagram of the TG framework.

The class diagram of the TG framework is showed in Figure 3.2. Each components of the TG framework is introduced as follows.

- A Game_Object represents a physical object in the tabletop games. It is identified by the attribute obj_id. As the definition in the TG model, a Game_Object has one or more faces and the attributes x, y, z, and f indicate the Game_Object's X-coordinate, Y-coordinate, Z-order, and current face respectively. Note that the unit of the X and Y-coordinate is pixel.

- A Game_Area is a two-dimensional area representing the game table or a private area. It is identified by the attribute area_id. A Game_Area contains zero or more Game_Objects while each Game_Object must be placed in one and only one Game_Area.

  In order to simplify the implementation, each Game_Area has finite size and is disjoined with other ones if there are two or more Game_Areas in a tabletop game. Therefore, it is easy to draw all Game_Areas in a two-dimensional space. Note that the unit of position and size of a Game_Area is pixel.

- The Game_Operation_Actor interface provides a means that allows a Game_Status to control the TG framework. A Game_Status uses this interface to define Game_Objects and Game_Areas, to move a Game_Object (within one Game_Area or between two), to maximize a Game_Object's Z-order, and to change a Game_Object to the next one.

- The Game_Operation_Checker interface provide a means that allows the TG framework to report a player's operations to the Game_Status. The reported operations are the three defined in the TG model. The Game_Status should implement this interface to receive a player's operations.

- The Game_GUI object plays the manager role in the TG framework. It contains one more Game_Areas and is responsible to draw all Game_Areas and Game_Objects on a Java AWT Canvas object. It draw Game_Areas and Game_Objects in the way specified in the TG model. Besides, Game_GUI implements the Game_Operation_Actor so that the Game_Status can define and control Game_Areas and Game_Objects.

  On the other hand, Game_GUI receives a player's actions such as mouse clicks or drag-and-drops on the Canvas object and converts them into the operations on the Game_Objects. Then, it reports the operations to Game_Status through the Game_Operation_Checker interface.

  Currently, Game_GUI receives the mouse left button click, right button click, and drag-and-drop actions and converts them into the max-Z, next-face, and move-object operations on the Game_Objects respectively.

3.3 A Gobang Based on the TG Framework

In this section, a Gobang is presented to demonstrate how to develop a game based on

the TG framework. The demonstration focuses on using the TG framework's API to control the GUI of the Gobang game.

Gobang is a two-player board game. The objects used in a Gobang game include 213 black pieces, 212 white pieces and a board with 15x15 intersections. One player uses black pieces and the other uses white pieces. They play the game by placing one of their pieces at an empty intersection in turn. The player who uses black pieces plays first. The winner is the first one who gets an unbroken line of five pieces horizontally, vertically, or diagonally.

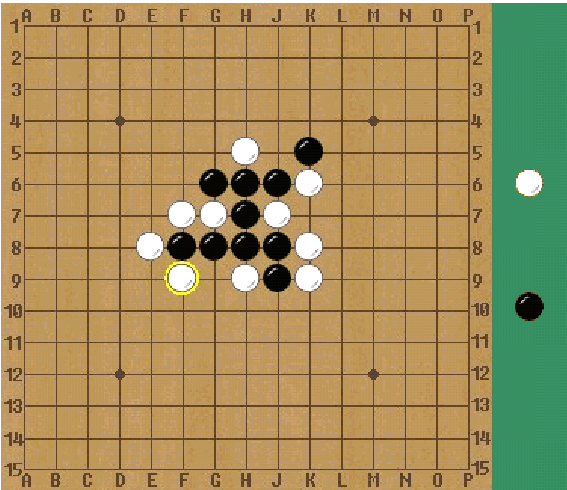


Figure 3.3. A screenshot of the TG framework based Gobang.

Figure 3.3 shows a screenshot of the TG framework based Gobang. Note that there are a pile of black pieces and a pile of white pieces near the right side of the board. The players use their mouses to drag a black or white piece from the pile of the black or white pieces and then drop the piece on an empty intersection on the board. The reason why using the mouse drag-and-drop action is to simplify the implementation since the action is built-in the TG framework.

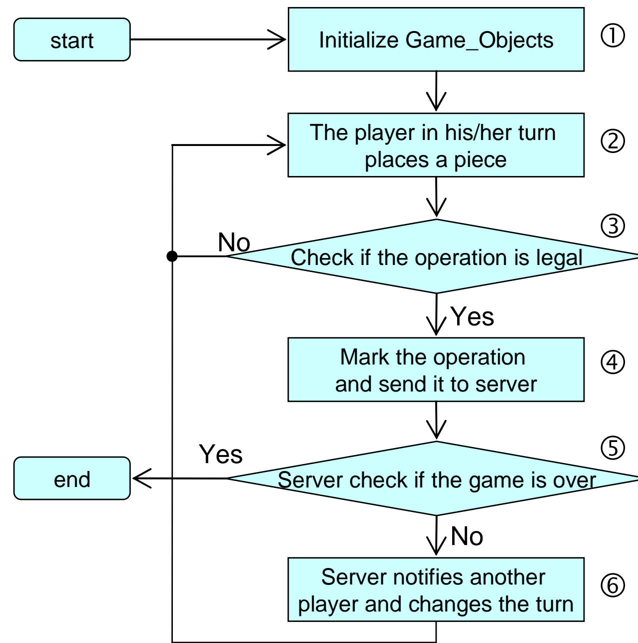


Figure 3.4. The main flowchart of the TG framework based Gobang.

Figure 3.4 shows the main flowchart of the Gobang. It describes the major steps of a round of the game. Before the first round starts, the `Game_Area` and `Game_Objects` (the black pieces, white pieces and the board) should be defined with the `define_Game_Area` and `define_Game_Object` methods of the `Game_Operation_Actor` interface. Each step in Figure 3.4 is described as follows.

- In step ①, all black and white pieces are moved to the position of the black and white piece pile by the `move` method of the `Game_Operation_Actor` interface.

- In step ②, the player in his or her turn uses the mouse drag-and-drop action to place a piece on an empty intersection on the board. The TG framework handles this move operation and then invokes the `move` method of the `Game_Operation_Checker` interface to inform the `Game_Status` of the Gobang.

```
Game_Status implements Game_Operation_Checker
{
  int board[][] = new int[15][15] ;
  Game_Operation_Actor actor;

  boolean move(int obj_id, int area_id, int x, int y) {
    if(!isMyTurn()) return false;          // ③
    Point p = toLogicalCoordinate(x, y);   // ③
    if(!isEmpty(p.x, p.y)) return false;   // ③
    board[p.x][p.y] = obj_id;              // ④
    sendToServer(obj_id, p.x, p.y);        // ④
    return true;
  }

  void refresh_move(int obj_id, int area_id, int x, int y) {
    Point p = toPhysicalCoordinate(x, y);  // ⑥
    actor.move(obj_id, area_id, p.x, p.y); // ⑥
    changeToNextTurn();                    // ⑥
  }

  ...
}
```

Figure 3.5. Part of the TG framework based Gobang program.

- In step ③, the `Game_Status`'s `move` method is invoked to check if the move object operation is legal. The method is showed in Figure 3.5. The method first checks if the turn is the player's turn. Then the method checks if the piece is placed in an empty intersection by looking up the `board` attribute. If the move object operation is illegal, the `move` method returns false and the TG framework will cancel the move object operation by moving the moved object to its original position.

  Note that since the TG framework measures the object positions in pixels, the x and y parameters of the `move` method should be translated into the logical coordinates used in the `Game_Status` and the game server.

- In step ④, the move object operation is marked in the `board` attribute of the `Game_Status` for next checking. Then the operation is sent to the game server.

- In step ⑤, the game server checks if the round of the game is over and decides which player is the winner. If the round is over, some messages should be showed to tell the players who is the winner. Then the players can decide to quit the game or play another round.

- In step ⑥, the game server notifies another player who is named player B below the move object operation. Player B receives this notification and the `refresh_move` method of player B's `Game_Status` is invoked. In the method, the `move` method of the `Game_Operation_Actor` interface is invoked to inform the TG framework the about move object operation so that the TG framework moves the object to the specified position.

  Then, player B gets the turn of the game and the game continues.

3.4 An Universal Tabletop Game System

From Theorem 2.4 in Subsection 2.2.2, we know that both TG game systems Ψ^*_C and Ψ^*_{MMN} are general, that is, for all TG games *G*, there exists some game *G*' in Ψ^*_C and Ψ^*_{MMN} that simulates *G*. In this section, we practically implement a TG game system Ψ^*_{MMN} based on the TG framework to support all TG games openly. Note that we are more interested in Ψ^*_{MMN} than Ψ^*_C because it is much easier to implement the three operations, move-object, max- Z, and next-face as we will see in the remaining of this section.

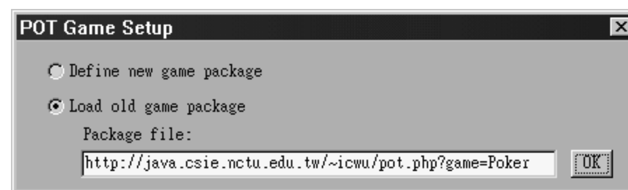


Figure 3.6. Initializing a game.

From Theorem 2.4, for all TG games *G*, there exists some game *G*' in Ψ^*_{MMN} that simulates *G*. More specifically, if *G* is a *p*-TG game and $G = (A^{(p+1)}, O, S, M)$, then $G' = ((R^2)^{(p+1)}, O, S^{(p)}{}_C(O), M^{(p)}{}_{MMN}(O))$. Obviously, the only parameters for *G*' are *p* and *O*. Thus, in our real implementation, the game system allows players to set the numbers of private areas

and all the game objects initially. For simplicity, we assume that one player uses a distinct private area in the rest of this section. This implies that the number of players are the same as that of private areas.

In order to facilitate players' operations, our game system allows players to choose to set the player numbers and the game objects via game packages that include these data. In order to retrieve the game package more easily, players only need to specify the URL location of the game package, as shown in Figure 3.6.

Alternatively, players can choose to set the player numbers and the game objects via game packages manually as shown in Figure 3.7. When adding new game objects, players specify the objects' names, initial positions (including the area index, x, and y), and faces (a list of URLs to the image files of faces, separated with commas), and then click on the button "Add/Modify". Players can choose to update an object by clicking the row of the object. When all objects are set, players can choose to save the package by clicking on the button "Save", start playing by clicking on the button "Play", or leave the game by clicking on the button "Quit".

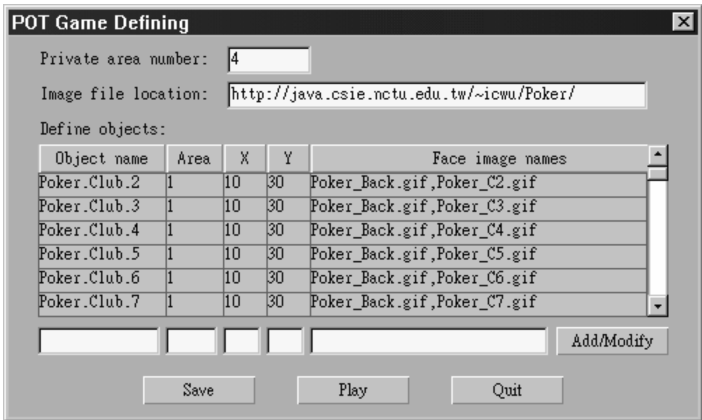


Figure 3.7. Defining a new TG game.

After players start playing the game (by clicking on "Play"), players can only see the public window or their own private windows (note that each window is corresponding to one

area). For any other windows, players cannot see. However, in our implementation, an option is given to choose to see the back face of objects in the private window and the back face is set to be the first face in the face list by default. Figure 3.8 illustrates the game appearance of player one.

Since Theorem 2.4 shows that it is general for the game system to support the move-object, max-Z, and next-face operations, our physical game system supports the three kinds of operations in the following way:

- The move-object operations: Use traditional drag-and-drop actions to change locations of targeted objects.
- The max-Z operations: Players can click on targeted objects to raise the Z-order of the objects with the *left* mouse button.
- The next-face operations: Click on targeted objects to change the faces of these objects to the next ones (in the lists of faces) with the *right* mouse button.

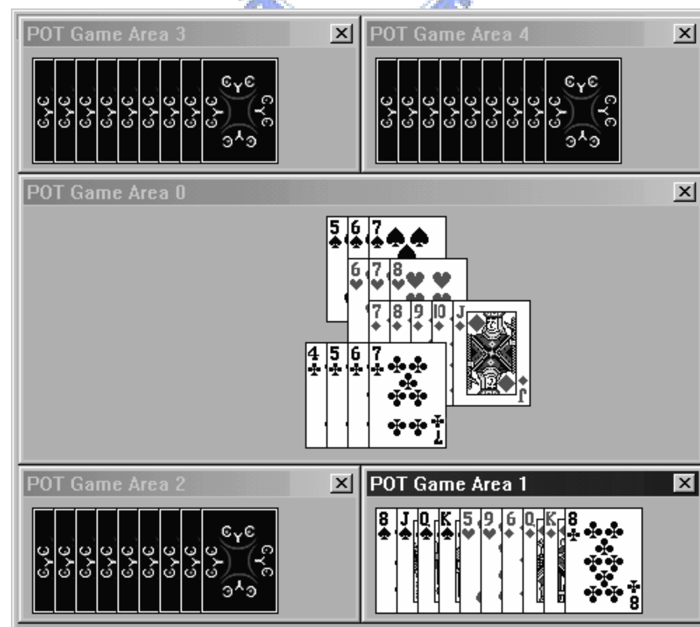


Figure 3.8. A game appearance for player one.

Since all the basic operations in the TG game system Ψ^{*}_{MMN} are implemented, it is

sufficient for players to play (or simulate) all TG games in the physical game system as described above.

Chapter 4 Related Issue 1: Resolving Problems of Blocking I/O Operations for Server Programming

When design and implement game servers, the use of blocking I/O operations is straightforward but may freeze the game servers in some cases so that the services of the server stop for an unexpected period of time. In this chapter, we discuss this issue in detail and present our solution. Since the issue affects not only game servers but also other applications involving real-time communication among clients, we use the term *inter-user communication applications* to refer to these types of applications in the dissertation.

4.1 Server Programming for Inter-User Communication

With the rapid growth of the Internet, applications involving real-time communication among clients have become increasingly important. Consider an example of chat room or game system. One user types a message and others then can read that message in real time. Since these applications involve inter-user communication, this dissertation calls them inter-user communication applications.

For inter-user communication applications, servers are often used to handle inter-user communication. For example, game servers receive player events (or messages) and then respond (or pass messages) to other players. For inter-user communication applications, server developers generally must consider the following criteria.

1. Minimize the client response time. If the response time is unexpectedly long, interactions may not evolve as expected or users may run out of patience.
2. Ensure high server stability. Server crashes cause all clients connected to that server to become disconnected.

3. Support as many clients concurrently as possible. For example, support thousands of players on a single server.

The first criterion is essential for server programming in inter-user communication applications. To respond to users as rapidly as possible, servers usually hold connections to clients. Servers thus must handle client messages (or events) from all connections concurrently and server developers must handle concurrent events carefully.

Two main programming models exist for concurrent event handling, namely threading and event-driven programming. Threading is a general-purpose technique for managing concurrency. The advantages of threading compared to event-driven programming include: (a) support of context switching among threads, and (b) support of scalable performance on multiple CPUs.

However, some developers and researchers [57, 66] also have observed that threading has some drawbacks compared to event-driven programming. Note that Ousterhout [57] described the following drawbacks:

1. Difficult to program. Threads generally require synchronous mechanisms (e.g., locks) to access shared data safely. However, incorrect locking may cause deadlocks, making independent module design difficult. Besides, another problem that also increases programming difficulty is that several standard libraries are not thread-safe [51].
2. Hard to debug. For threading, it is difficult for developers to debug the code due to data and timing dependencies. Besides, another problem that also increases debugging difficulty is that thread stack sizes are normally limited [46, 51], causing processes crash when stacks overflow. In contrast, in event-driven programming, the lack of context switching among event handlers makes it quite easy to debug the code by recording and then replaying the sequence of events.

3. Difficult to achieve good performance. Coarse-grain locking yields low concurrency, while fine-grain locking tends to increase lock operations and thus reduce performance.

Since inter-user communication applications are often used to facilitate heavy inter-user communication among numerous clients (say, thousands of players in a game system), it makes the above drawbacks even worse. The first two drawbacks imply that it is hard for threading to satisfy the second criterion (above) of the inter-user communication applications; and the third drawback indicates that it is hard for threading to satisfy the third criterion. Thus, for the application developers who are more concerned with the second and third criteria and less concerned with the two threading advantages (described above), the event-driven programming model becomes attractive. Hence, this dissertation is motivated to study and design an event-driven framework for inter-user communication applications.

Our framework is based on event-driven programming (rather than threading) for the following reason. In inter-user communication applications, the above three drawbacks of threading (or the second and third criteria) are important as described above, while the two drawbacks of event-driven programming are less important because they can be ignored or alternatively can be solved in this dissertation. First, regarding the two drawbacks of event-driven programming, this dissertation ignores the one, namely not supporting scalable performance on multiple CPUs, because for most inter-user communication applications servers can be separated into several processes to achieve scalable performance. In the case of tabletop games, such as Chess and Bridge, servers can naturally be separated into several processes, e.g., one for each game. Even for most massive multiplayer online games (MMOGs), such as Ultima Online [22], the server system can use several processes each dealing with a single game scene. Second, this dissertation focuses on overcoming the other drawback of event-driven programming: the need to pay attention to the blocking problems due to the use of blocking I/O operations in event handling.

This dissertation addresses two major blocking problems and presents solutions or guidelines. The two blocking problems are described below.

1. Output blocking: This problem occurs on sending messages to clients with corresponding full kernel buffers. The buffer generally becomes full when network traffic is jammed. This problem frequently is neglected at the start of server development.
2. Request blocking: This problem occurs when a server waits for responses after sending requests to other servers. For example, when a game server attempts to read several game records from a remote database server.

This chapter presents solutions for the above two blocking problems. An output buffering mechanism is presented to solve the output blocking problem, while a service brokering mechanism is presented to solve the request blocking problem. Meanwhile, for the second problem, several system and library calls that may cause the problem are also identified. Both mechanisms are incorporated into the event-driven framework presented in this chapter.

The rest of this chapter is organized as follows. Section 4.2 reviews the event-driven programming model. Section 4.3 describes the output blocking problem and presents solutions. Section 4.4 then describes the request blocking problem and presents solutions. Section 4.5 presents performance analysis.

4.2 The Event-Driven Programming Model

This section reviews the event-driven programming model. In this model, applications wait for specific events and dispatch occurring events to appropriate handlers for processing. In networked applications, event-driven based servers generally handle both input and output events. Input events occur when sockets are ready to read, while output events occur when sockets are ready to write.

Most event-driven based servers in the Unix environment use the `select` system call [19, 41, 72, 74] to demultiplex input/output events. The `select`-based event-driven model has been induced as the Reactor design patterns in [65, 66]. In this pattern, the core component named *`Reactor`* waits for input/output events synchronously. When such events occur, the `Reactor` object identifies the handlers of these events and then invokes the appropriate methods of the handlers.

The Reactor pattern defines an event handler interface. Concrete event handlers implement the event handler interface to support application specific services. These concrete event handlers are registered with the `Reactor` object dynamically, and then are passively reacted to the occurrences of designated events. Application developers only need to implement concrete event handlers, when reusing the dispatching mechanism of the `Reactor` object.

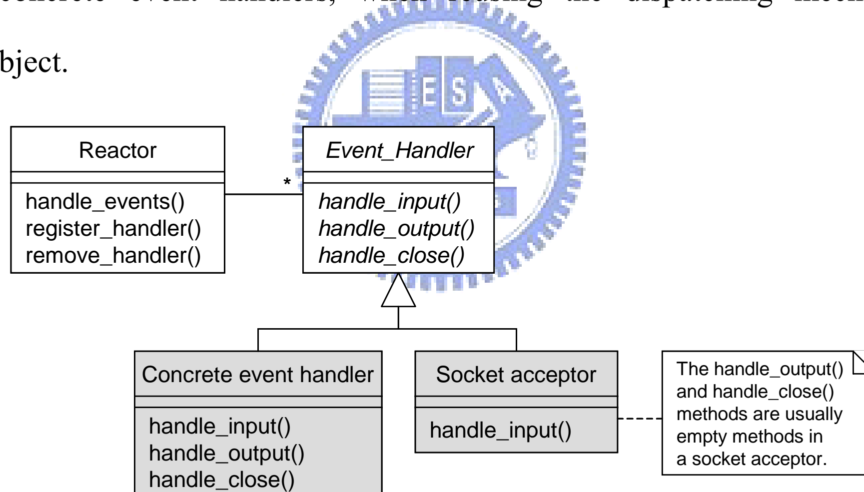


Figure 4.1. Class diagram of the Reactor pattern.

Figure 4.1 shows the class diagram of the Reactor pattern. Note that the shadowed classes are application-specific (that is, application developers must implement these classes only). The responsibilities of each class are detailed as follows.

- `Reactor` is the core component of event-driven applications and a process normally requires only one `Reactor` object. Applications can register and remove event handlers by calling `register_handler` and `remove_handler` of this object respectively.

The handle_events method in this object is invoked to run the event-handling loop, in which the select system call is used to wait for some specified input or output events synchronously. As events occur, the Reactor object dispatches those events to the corresponding event handlers.

- Event_Handler is an abstract event handler class that defines several hook methods [29, 60]: handle_input, handle_output, and handle_close. Concrete event handlers are application specific event handlers that inherit the Event_Handler class. When an input (or output) event occurs for some concrete event handler, the handle_input (or handle_output) method of the handler is invoked to process the event. Before handler removal, the handle_close method is invoked for application specific termination operations.
- A socket acceptor is a special concrete event handler that is responsible for accepting new connections. The handle_input method of the socket acceptor accepts a new connection from a client (by calling accept in Unix), generates the corresponding concrete event handler for the client and then registers this handler with the Reactor object. Since a socket acceptor does not output data and hold state, its handle_output and handle_close are generally empty.

The above Reactor pattern forms a basis of the event-driven framework presented in the remaining part of this chapter.

4.3 Output Blocking Problem and Solution

This section discusses the output blocking problem for event-driven programming in inter-user communication applications. Subsection 4.3.1 describes the output blocking problem, and Subsection 4.3.2 then presents a mechanism for solving this problem.

4.3.1 Output Blocking Problem

In most TCP/IP implementations, each socket contains both send and receive buffers [19, 74] in the kernel. Both buffers are tens of kilobytes in size in Unix. When the send buffer of a socket in the kernel is full, output to the socket is blocked, since by default all sockets are in the blocking mode [74].

Output blocking is a serious problem in inter-user communication applications. For example, in a game system, it is common for a server to receive one message from one player, say *A*, and then immediately send that message to a set of players, say including player *B*. However, if network traffic is jammed near or around player *B* (but not elsewhere), the server blocks due to the failure to send the buffer of the socket to *B*.

From our experience with the CYC Game League [82], the output blocking problem seriously degrades the performances of inter-user communication applications, since network traffic may be jammed unexpectedly. A more serious situation is the following: if a client crashes or its network wire is disconnected, the server may not detect the disconnection by default for approximately nine minutes [73]. Furthermore, since a server for a game system usually serves thousands of players or more, it is easy to cause server blocking as above and result in slow responses to all clients.

4.3.2 Solution to the Output Blocking Problem

Stevens (cf. Section 15.2 of [74]) demonstrated a simple buffering method when discussing nonblocking I/O. Since the buffer size is fixed to a small number, this method still cannot solve the output blocking problem when the message size is larger than the buffer size. Some event-driven based web servers, such as thttpd [1] and mathopd [11], used the `sendfile` system call [83] as well as nonblocking sockets to avoid the output blocking problem upon sending files. The Flash web server [58] also proposed a method that can avoid

the output blocking problem for sending web pages. The above work solved the problem specifically for their own applications. In this chapter, we propose a reusable framework for generally solving this problem.

In order to solve this problem in event-driven based servers, this dissertation presents a mechanism, called an *output buffering mechanism*, and incorporates it into our event-driven framework. This mechanism sets all the sockets to the non-blocking mode and extends event handlers to those with extra dynamic output buffers. The buffers are used to store unsent data that cannot be sent when the send buffers of sockets are full, as described above. Namely, the unsent data are stored into the extra output buffer when the socket send buffers are full, and the buffered data then are sent out whenever the send buffers have available space.

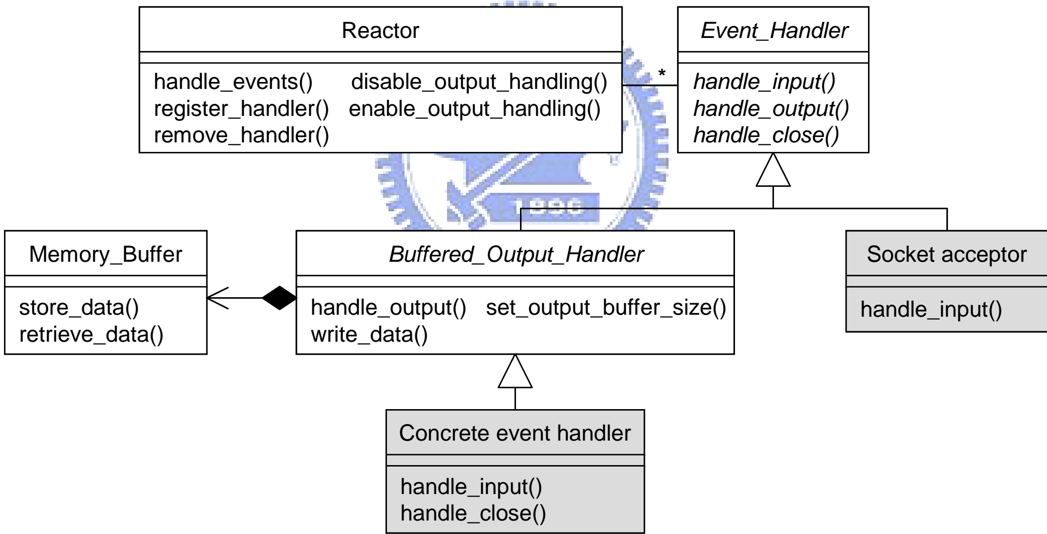


Figure 4.2. Class diagram of the event-driven framework with the output buffering mechanism.

Figure 4.2 shows the class diagram of the event-driven framework with the output buffering mechanism. This dissertation simply describes the extra classes and methods in this Figure, when compared to Figure 4.1, as follows.

- `disable_output_handling` and `enable_output_handling` are two new methods added to the `Reactor` class. The former method disables handlers from

handling output events, while the latter method enables handlers to handle output events. Output handling is initially disabled for all event handlers.

- `Buffered_Output_Handler` is an abstract class that partially implements the `Event_Handler` interface. Specifically, `Buffered_Output_Handler` uses the `handle_output` method to handle unsent data, while leaving the two methods `handle_input` and `handle_close` unimplemented. The classes of concrete event handlers extend the class `Buffered_Output_Handler`, rather than `Event_Handler`, and only need to implement the above two unimplemented methods.

  In order to hide output handling from application developers, the `Buffered_Output_Handler` class hides the method `handle_output` and provides developers with the `write_data` method, rather than the `write` system call. The `write_data` method normally writes data out as the `write` system call, but stores the unsent data into a buffer on output blocking and thus enables output event handling.

- `Memory_Buffer` is a class of dynamic sized buffers.

  Each concrete event handler allocates a single `Memory_Buffer` object to store unsent data when the send buffer of the corresponding socket of the handler is full. Since send buffers in the kernel rarely become full, the physical buffer spaces of the dynamic output buffers are created only when required and are immediately freed when not required. Normally, the maximum buffer size is set to a large number, for example 1M bytes. Since each message in the inter-user communication applications is generally small, overflowing of `Memory_Buffer` usually implies that the network has been jammed for a while. Thus, it is reasonable to claim connection failure in such situations.

This dissertation now illustrates the following interactions in more detail, including: (1) how to accept a new client, and (2) how to handle output buffering. Note that the UML sequence diagram [12] is used to demonstrate these interactions.

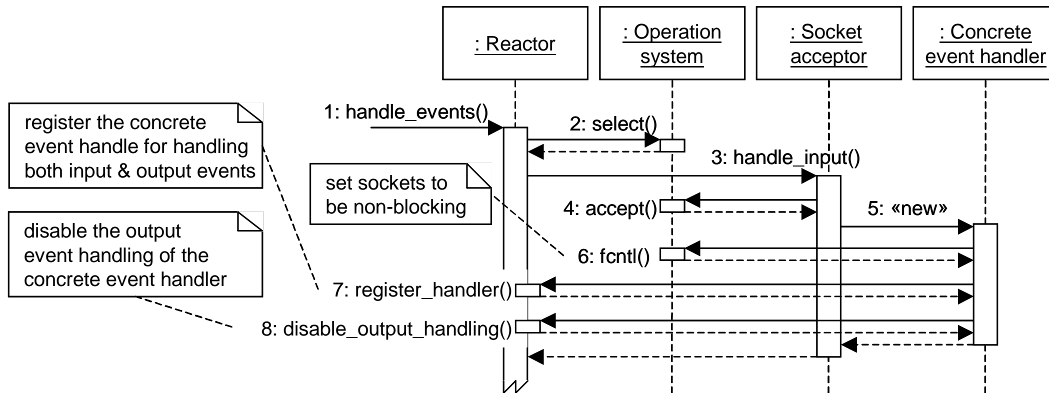


Figure 4.3. Sequence diagram for accepting a new client.

First, the sequence diagram in Figure 4.3 illustrates how the socket acceptor accepts a new client. When invoked to handle an input message (in Step 3), the socket acceptor accepts a connection request (in Step 4) and creates a concrete event handler (in Step 5). The concrete event handler then sets the corresponding socket to the non-blocking mode (in Step 6), registers itself for event handling with the `Rector` object (in Step 7) and initially disables output handling (in Step 8).

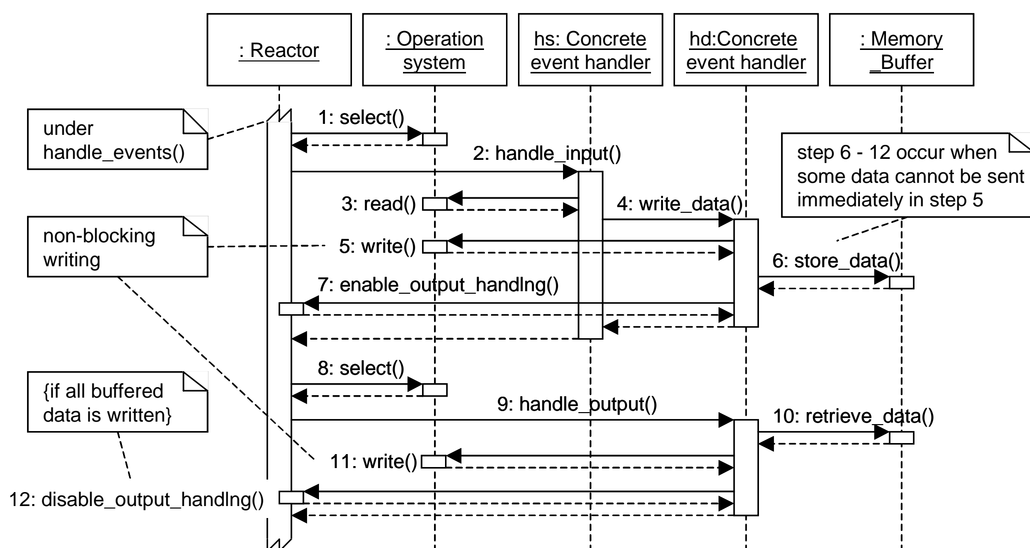


Figure 4.4. Sequence diagram for handling output buffering.

Second, the sequence diagram in Figure 4.4 illustrates how a concrete event handler, `hs`, writes messages to another handler, `hd`, with a full send buffer while handling input messages. In this case, the messages (from `hs`) cannot be sent out due to the send buffer of the corresponding socket in `hd` being full (in Step 5). Subsequently, in `hd`, the unsent data is stored into its own `Memory_Buffer` (in Step 6) and output handling is enabled (in Step 7) to output the unsent data later. When the socket has available space for output, the `handle_output` method of `hd` (in Step 9) is invoked to send the data in `Memory_Buffer` out (in Steps 10 and 11). Finally, output handling for `hd` (in Step 12) is disabled if all the data are sent out successfully.

4.4 Request Blocking Problem and Solution

This section investigates the request blocking problem for event-driven programming in inter-user communication applications. Subsection 4.4.1 introduces the request blocking problem. Subsection 4.4.2 then presents a solution to solve the request blocking problem for HTTP requests only. Next, Subsection 4.4.3 leverages the solution in Subsection 4.4.2 to solve the request blocking problem for all other requests.

4.4.1 Request Blocking Problem

For event-driven programming, application developers must also be careful about using possible blocking operations in event handling. Besides the output blocking operations in Section 3.3, blocking operations are classified into two types: namely local blocking operations and request blocking operations.

The former include explicit system or function calls that may block execution locally, such as `wait`, `sleep`, `flock`, and `semop` [72]. Developers should either prevent from using these functions or use alternatives, instead.

Request blocking operations are involved in service requests over network. For example, when a game server S_G needs to retrieve game records from (or store records into) a database server S_{DB}, S_G generally performs the following three steps: (1) create a connection to S_{DB}; (2) send request messages to S_{DB}; (3) receive response messages from S_{DB}. However, the operation in step (3) is clearly a blocking one.

In event-driven based servers, if a straightforward design is used that directly bundles the three operations together within a single input event handler, called a *source event handler* here, this service request obviously becomes blocked. Consequently, the performance of the game server S_G degrades.

Many developers usually notice the above example for database requests before coding. However, unfortunately many services are requested implicitly. For example, the Harvest and Squid projects [13, 90] noticed that the DNS-related function call, `gethostbyname`, may issue a request to a DNS server and wait for the response from that server. Thus, for event-driven programming, it becomes crucial for application developers to identify more operations with remote service requests, as listed below.

- DNS-related functions. For example, accessing DNS servers via some library calls such as `gethostbyname`, `gethostbyaddr`, `getaddrinfo`, and `getnameinfo` [19].
- Database-related functions. For example, accessing database servers via JDBC [26, 76] or ODBC [45] drivers.
- LDAP-related functions. For example, accessing the servers of OpenLDAP via its client library [54].
- HTTP access. For example, making HTTP requests via `libwww` [52].
- Remote file access. For example, accessing a file mounted on a remote host via the NFS service [18, 19].

Note that the last operation involving remote file access may also block event-driven based servers for inter-user communication applications, because traffic to the remote file server may also be jammed.

Regarding local file access, the research in [58] indicates that most local file operations generally cannot be integrated with the `select` system call. Namely, `select` cannot be used to detect the completion of these operations. Furthermore, some of these operations, such as `open` and `stat`, may still be blocking. The above blocking problem is critical for the HTTP server applications [58] because HTTP servers generally require frequent accessing of local files. However, this dissertation is less concerned with local file access, since inter-user communication applications generally process messages on the fly without frequently accessing local files. For example, a game server generally does not need to save player chat messages into local files. If a game server does need to access files frequently for some reason, the server can use database, instead, and the solution presented in the remainder of this section remains useful.

In order to solve the request blocking problem in event-driven applications, the Harvest and Squid projects [13, 90] used helper processes to resolve DNS queries without incurring blocking. However, they did not design a reusable software architecture for this problem.

For solving this problem in a reusable way, this dissertation first presents a service brokering mechanism for dealing with HTTP requests in Subsection 4.4.2. Then, in Subsection 4.4.3, this mechanism is applied to all the other service requests with blocking operations.

4.4.2 Solutions for HTTP Access Requests

This subsection presents a mechanism, called the *service brokering mechanism*, for dealing with HTTP requests, and incorporates this mechanism into the event-driven

framework in this dissertation. In this mechanism, an event handler, called the *source event handler*, creates another event handler, called the *service broker* here, to send an HTTP request to a remote service provider and wait for the response. After receiving the response, the service broker transfers the response back to its source event handler. These activities are performed without any blocking.

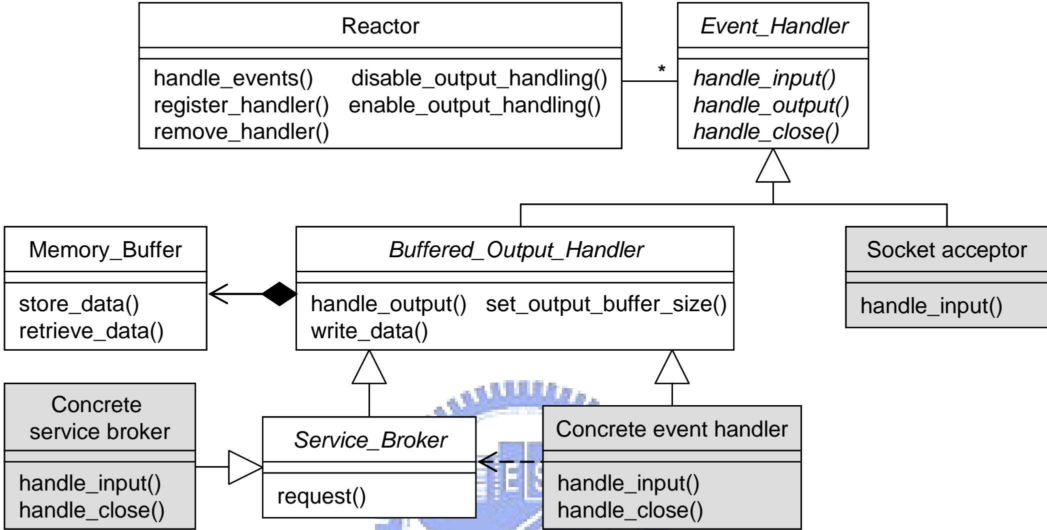


Figure 4.5. Class diagram of the event-driven framework with service brokers.

Figure 4.5 modifies the class diagram of the framework in Figure 4.2 by adding two classes, `Service_Broker` and concrete service broker, detailed below.

- `Service_Broker` is an abstract class that extends `Buffered_Output_Handler` for service brokering.

  This class provides application developers with a `request` method that is used to establish a connection to a server, such as an HTTP server, and forward an HTTP request to the server. A concrete event handler (or the source event handler as defined in Subsection 4.4.1) requires the following parameters to invoke this method: (1) server IP address and port; (2) the pointer back to the source event handler; and (3) the HTTP request message.

- Concrete service brokers are application-specific classes implementing the `Service_Broker`.

  A concrete service broker object is created by a source event handler to handle one and only one HTTP request. After requestor creation, the source event handler invokes the `request` method of the requestor to connect to the corresponding HTTP server and then registers the requestor with the `Reactor` object. When the server replies, the `Reactor` object invokes the `handle_input` of the requestor to process the response.

Next, the following interactions are illustrated in more detail: (1) how to establish a connection to a remote service provider and send an HTTP request to that provider, and (2) how to handle service provider responses.

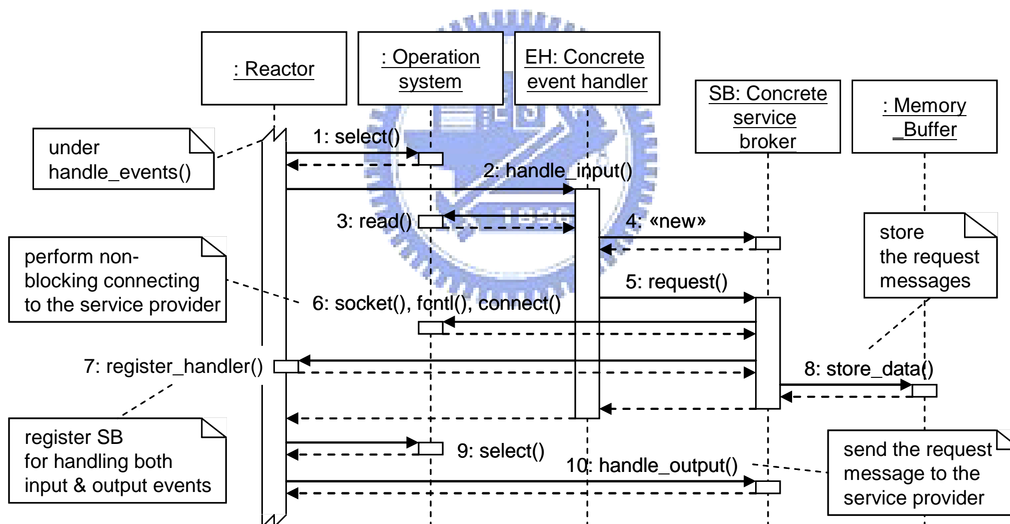


Figure 4.6. Sequence diagram for establishing a connection and sending an HTTP request.

First, the sequence diagram in Figure 4.6 illustrates how a concrete event handler, `EH`, establishes a connection to a remote service provider, `SVCP`, and forwards an HTTP request to that provider. When making an HTTP request, `EH` creates a service broker, `SB`, (in Step 4) and then call the `request` method of `SB` (in Step 5). This method connects to `SVCP` in a non-blocking manner (in Step 6), registers `SB` itself with the `Reactor` object (in Step 7) and

stores the request message in the `Memory_Buffer` (in Step 8). The stored request message is sent (in Step 10) immediately upon connection establishment.

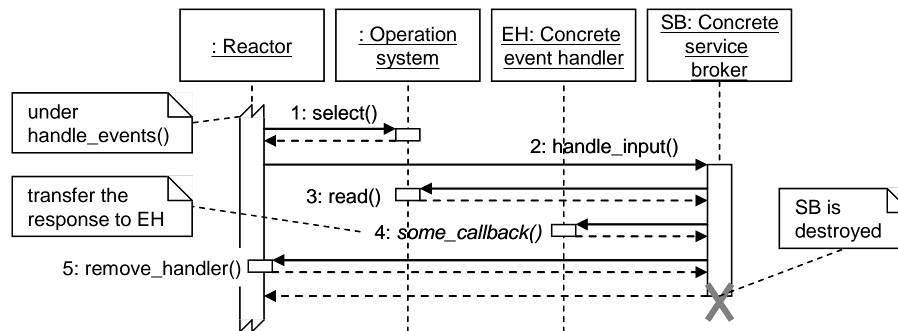


Figure 4.7. Sequence diagram for handling responses.

Second, the sequence diagram in Figure 4.7 illustrates how the service broker, `SB`, handles the responses from the service provider, `SVCP`. On receiving responses from `SVCP` (in Steps 2 and 3), `SB` processes those responses and may invoke some callback functions of `EH` (in Step 4). Once the response has been completely received, `SB` destroys itself (in Step 5).

4.4.3 Solutions for Other Service Requests

The previous subsection presents a service brokering mechanism for dealing with HTTP requests in the event-driven programming model. Since HTTP and its tools are pervasive, for example web servers like Apache [78] and IIS [44], and web programming tools like PHP [80], JSP [75] and ASP [43], a straightforward solution for all the service requests would be to leverage the above solution based on the HTTP servers directly.

For example, if a source event handler (defined in Subsection 4.4.1) needs to access database servers or get the IP address of a given hostname, the handler makes an HTTP request to a web server, and the corresponding web page programs (say in PHP) then return database records or the IP address. Since it is easy to write the code of service brokers, as described in Subsection 4.4.2, and the corresponding web page programs (in PHP, JSP, or ASP), application developers can easily develop the above service request. Note that web

page languages such as PHP, JSP, or ASP are usually sufficiently general and high-level to program service requests such as those listed in Subsection 4.4.1.

However, leveraging the HTTP technologies as above may incur significant overhead for web page processing (in PHP, JSP, or ASP). For example, accessing a database or getting the IP address of a given hostname in PHP generally may include process forking and page interpretation.

Since the incurred overhead may become significant, this dissertation designs additional helper processes for handling requests directly. The idea of helper processes has been used by the researchers in [13, 58, 90] for calling DNS-related functions and disk I/O access. However, they do not design a reusable software architecture for helper processes, as this dissertation does.

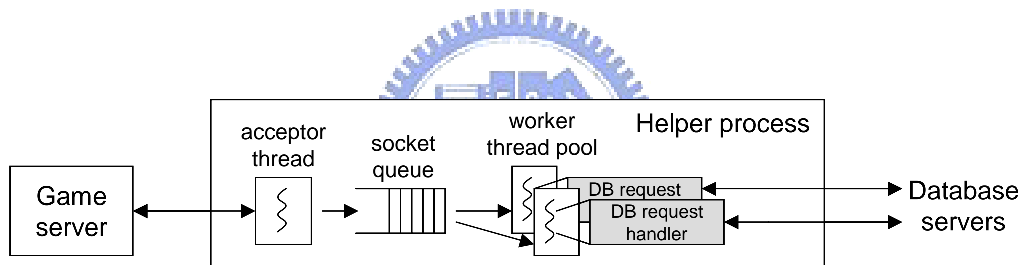


Figure 4.8. Handling database requests using multi-threads.

Consider the example of accessing database servers. A helper process contains an acceptor thread and a pool of worker threads, as illustrated in Figure 4.8. The acceptor thread repeats to accept new connection requests from application servers (such as game servers) and then queues the sockets corresponding to these connection requests. Each worker thread then repeats the following steps:

1. Retrieve one socket from the queue.
2. Receive the request message (including the URL and the parameters) from the socket.
3. Identify the request and then create the corresponding service handler to process that request. For example, for a database request, the corresponding service handler sends the requests to the database servers, receives the response messages and then returns the

results to the game server. Meanwhile, for a DNS request, the corresponding service handler simply calls the DNS library and returns the results to the application servers.

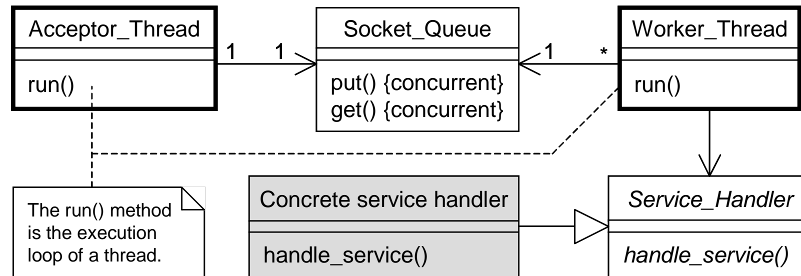


Figure 4.9. Class diagram of helper processes.

The class diagram in Figure 4.9 shows the components of the helper processes. The class responsibilities are described as follows.

- `Acceptor_Thread` is the thread that waits to accept incoming connections. The sockets corresponding to accepted connections are placed in a `Socket_Queue` (described below).
- `Socket_Queue` is the queue that stores socket descriptors.
- `Worker_Threads` are threads that process requests in the `Socket_Queue`. Each `Worker_Thread` object creates a concrete service handler (described below) for application specific services.
- `Service_Handler` is a class of service handler interface that defines a hook method: `handle_service`. Classes of concrete service handlers that inherit `Service_Handler` implement this method for application specific services, such as database access services.

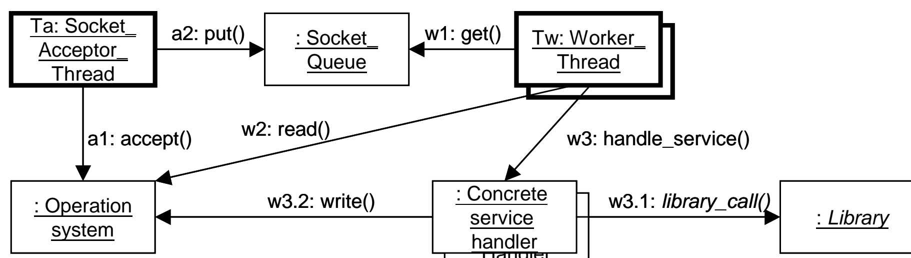


Figure 4.10. Collaboration diagram of handling services in a helper process.

Figure 4.10 shows the interaction of handling services in a helper process. The UML collaboration diagram is used to show this interaction.

- The acceptor thread `Ta` repeats the following two steps. In Step `a1`, it accepts a new socket, and in Step `a2`, it places accepted sockets in the `Socket_Queue`. If one or more worker threads are waiting in the queue, one of them is woken up.
- A worker thread, `Tw`, repeats the following steps. In Step `w1`, `Tw` attempts to obtain a socket from the `Socket_Queue`. `Tw` waits in this queue until the `Socket_Queue` becomes not empty. In Step `w2`, `Tw` reads the request message from the socket. In step `w3`, `Tw` invokes the `handle_service` method of its own concrete service handler by passing the request message as an argument. In Step `w3.1`, the method invokes library calls, such as `gethostbyname`, for the request. Finally, in Step `w3.2`, the method returns the result to the application server.

Note that the above helper process also has an additional advantage, solving the following problem, called the *limited service problem* in this dissertation. Consider that a database server generally supports a limited number of service connections. The problem can be easily solved in the helper process designed here by simply limiting the number of worker threads to the number of connections to the database server and letting each thread hold a single connection. For example, if a database server supports only 20 connections, the helper process dedicated to all requests to the database server has a maximum of 20 worker threads.

In fact, the advantage of the helper process described above also applies to the processing of HTTP requests, for the following reason. A web server such as Apache generally limits the number of daemon processes for simultaneous requests [78]. When the number of concurrent HTTP requests exceeds the limited number, some of the additional HTTP requests may fail to establish connections or suffer from long latency [89]. Consequently, the helper process can be used simply to limit the number of worker threads to

being the same as the number of HTTP daemon processes and thus let the service handler work like a HTTP proxy. For example, if an Apache server only allows 100 daemons, the helper process that is dedicated to all HTTP requests to the Apache server also has a maximum of 100 worker threads. The helper process thus can hold thousands of HTTP requests in the socket queue via the acceptor thread, while guaranteeing that 100 HTTP daemons in the Apache server are always available for the helper process.

The above thread pool can actually be implemented more efficiently, as described in [66, 89]. Briefly, when worker threads are idle, some of these threads can be dynamically removed to reduce the overhead of context switching. The details can be read in [89] and are omitted here.

From the above, this dissertation suggests that helper processes be deployed as follows. For each server with limited service resources (e.g., an Apache server with a limited number of daemons or a database server with a limited number of connections), one helper process is dedicated to the server. Meanwhile, other services such as DNS services can be grouped into one or more helper processes, depending on the situation. Figure 4.11 illustrates a case of deploying helper processes.

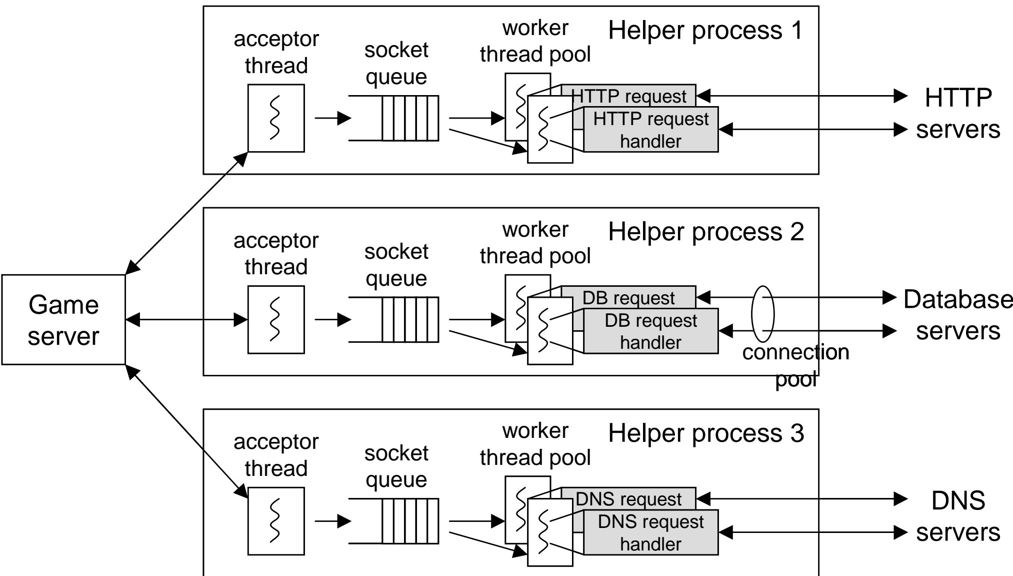


Figure 4.11. Case of deploying helper processes.

Finally, one may ask why the event-driven model is not used for the helper processes. Surely, the event-driven model can be used to implement the helper processes. However, the fact that these requests must wait for responses complicates helper process design. On the other hand, since threads in helper processes are almost independent, developers can easily maintain and debug the code in the thread model.

4.5 Experiments

This section presents the performance analysis for using the output buffering mechanism. The performance analysis for using the service brokering mechanism is similar and therefore is omitted in this dissertation.

From the CYC Game League, we logged all the events of some game server with about 300 players for 10 minutes. The log contains 28,573 received messages and 142,570 sent messages that represent the activity of the game server during that period.

For performance analysis, we simulated the activity of the log as follows. Let one host simulate the 300 clients (players) and the other simulate the game server following the messages indicated in the log. Both hosts ran on FreeBSD 5.3 and each of them was equipped with an AMD Athlon XP 2000+ CPU, 512 MB RAM, 80GB hard disk, and a 100 Mb Ethernet card. Besides, they were connected to a 100 Mb switch hub directly.

In our experiment, we only consider the response times of the messages among clients (players), e.g., the messages for chatting or playing cards. Namely, for each of such messages among clients, add into the message M the time when sending M from the sender. When receiving M, the recipient measures the traveling time of M from the sender. Normally, the response times are short, unless the server is overloaded or the network traffic is jammed or blocked.

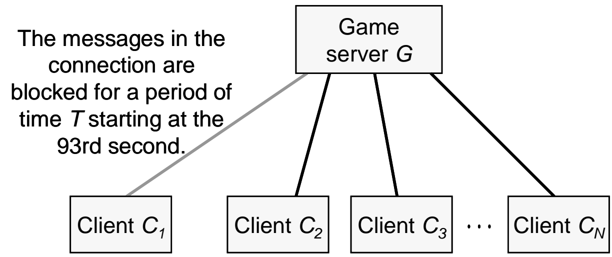


Figure 4.12. The deployment of the evaluating the output buffering mechanism.

From the log, we chose one client, called C_1, as shown in Figure 4.12, who entered the system at the 93rd second. Then, we blocked all messages to client C_1 for a period of time T starting at the 93rd second to simulate that the network between C_1 and the game server G was jammed or blocked, as shown in Figure 4.12. We used the technique of divert sockets [3, 17] to simulate the blocking network.

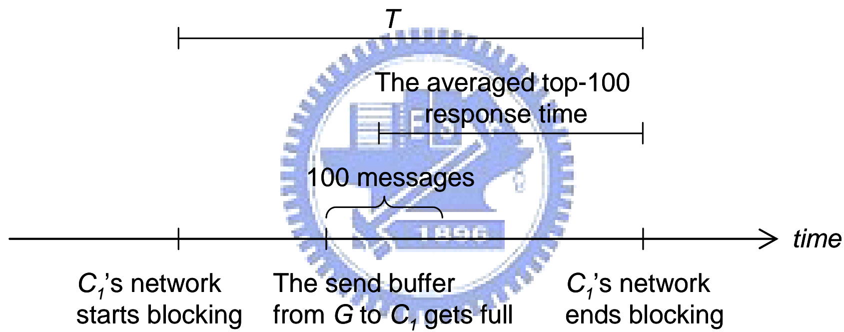


Figure 4.13. The averaged top-100 response time.

For each T, we measured the averaged top-100 response times in the two cases that (1) the output buffering mechanism was used and (2) the mechanism was not. Note that the averaged top-100 response time is the average of the highest 100 response times among all the messages. The top-100 response times reflect the worst response times among clients. In the case that T is sufficiently large, the averaged top-100 response time is the average of the response times of the first 100 messages after C_1's send buffer gets full, as shown in Figure 4.13.

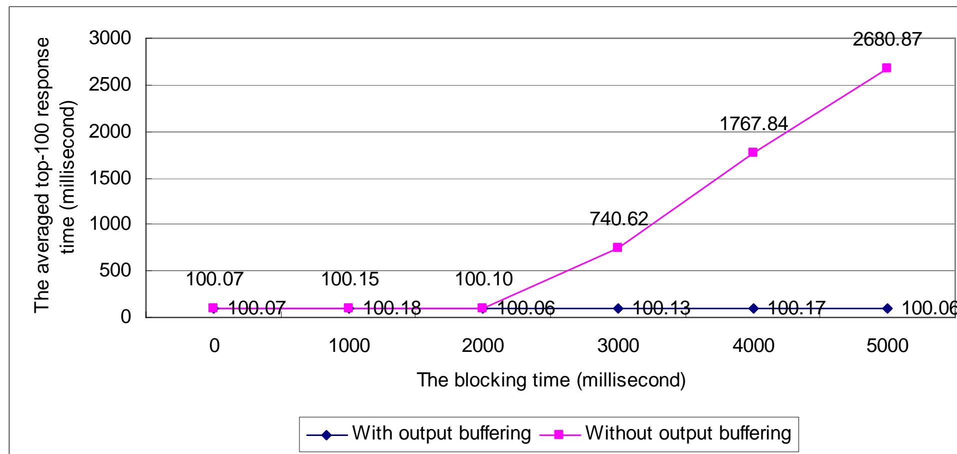


Figure 4.14. The averaged top-100 response times vs. the blocking times.

Figure 4.14 shows our experiment results. Consider the second case that the mechanism is not used. When $T \leq 2$ seconds, the averaged top-100 response times are still very low because in the FreeBSD kernel the send buffer (with about 32 kilo-bytes) from the game server to client C_1 is not overflown and the whole game system therefore is not blocked, except for client C_1. However, when $T > 2$ seconds, the send buffer is overflown and the whole game system is blocked for the rest period of time T. Thus, the averaged top-100 response times grow nearly linear as T in this case. If the output buffering mechanism is used, the averaged top-100 response times become all very low as shown in Figure 4.14. That is, the performance is greatly improved in this case.

E
S
A
1896

Chapter 5 Related Issue 2: Fast Recovery from Overflow Failures for Non-disruptive Game Services

The services of the GOTG system rely on the collaboration of several kinds of servers, including game servers, web servers, database severs, etc. However, the services may disrupt if some of these servers crash. To continuously provide the services, the servers should try to recover from failures timely. One of failures that we encountered when developing the GOTG system is the buffer overflow failure that is commonly found in many software recently. In the chapter, we develop a technique to detect and recover from buffer overflow failures efficiently.

5.1 The Buffer Overflow Problem

The C and C++ languages are popular in program development due to their efficiency and the capability of control over most resources, such as memory and I/O devices. However, many programmers can not correctly handle this capability of control and therefore introduce errors into applications. Among these errors, buffer overflow is a very common and serious error. According to our survey on SecuriTeam [6] and CVE [79], buffer overflows account for about 20% of errors in the recently years. The occurrence of the buffer overflows in a program generally cause the program to work incorrectly or even crash. Since many servers of the GOTG system are implemented in C/C++, we try to develop a technique to recover from buffer overflows.

A buffer overflow occurs when an array access is out of the bounds of the array. Since the C/C++ standard leave out of bounds(OOB) access undefined, most C/C++ compilers do not generate code to check this flaw. Although this can avoid the extra overhead of boundary checking, intruders can exploit buffer overflows to crash or intrude programs.

An OOB writing may overwrite language implementation structures such as function pointers and return addresses. The stack smashing attack [48] tries to overwrite a function return address to intrude a program. Once a stack smashing attack is successful, the victim program is hijacked and the intruder may get the superuser privilege.

Since buffer overflow is a serious vulnerability, many solution approaches have been proposed to tackle this problem. One way to defeat the buffer overflow vulnerability is by analyzing the source code of programs to discover the vulnerability [28][63][87][40]. However, source code analysis tools may generate considerable false positive warnings. Some researches combine source code analysis and runtime memory-error checking to discover memory errors [50][36]. These techniques can reduce the overhead of runtime checking but they are not fully compatible with the existing C-programs.

Another way to defeat buffer overflows is to check OOB accesses at runtime. Some techniques check only critical program status such as return addresses and previous stack frame pointers [20] [85] [15] [5] [23] and therefore only partially resolve the issues. Others check all OOB accesses [37][64][38][2][31] and therefore incur large runtime overhead in general.

Still another way is to randomizes the addresses of code and data [7][27][14][94]. These reported techniques can resolve most intrusions related to memory vulnerabilities without remarkable overhead since the intrusions rely on absolute addresses of data/code or relative distances among buffers. One major drawback of these buffer overflow prevention techniques is that they will terminate the victim programs upon being attacked. Although doing so can prevent intrusions, the termination of server programs may seriously affect the users of the servers.

In this dissertation, we propose a new technique, called BODAR (Buffer Overflow

Detection and Recovery), to recover from buffer overflows. Our method can avoid the above drawback and recover programs from buffer overflows and continue to work correctly.

5.2 The Design and Implementation of BODAR

In this section, we present the design and some important implementation issues of our buffer overflow detection and recovery method.

5.2.1 Integration of Address Space Permutation and Trapping OOB by Unallocated Address Space

Random address space permutation is a technique to prevent from out of bound (OOB) exploits by random arrangement of code, text, and stack segments. Emulation of non-executable stacks or data execution prevention without supports from memory management unit (MMU) also make use of similar permutation strategies. Any attempt to hijack controls of such random address space only results in failures of program execution, which suffers denial of services.

Electric Fence [59] is a replacement of the `malloc` family for debugging. It leverages virtual memory hardware to detect the accesses overrunning the boundaries of a buffer allocated with its `malloc`. Such an out-of-bounds access triggers a signal. Therefore when an Electric Fence enabled program runs under a debugger such as gdb, the faulty instruction can be displayed. However if the accesses jump over the range of the protected page (often 4-kbytes), the exception will not be triggered.

The Jones and Kelly (J&K) checker [37] proposed to handle out of bounds addresses (OOB) by checking each memory access to the OOB tag. If the access is an OOB dereference resulting from pointer arithmetic, the tag is replaced by a special ILLEGAL value. The CRED method replaces OOB value with an OOB object tracking referent objects by storing the OOB

value and the referent objects that the value refers to. The BMB (boundless memory blocks) method [62] allows OOB access for continuous execution with extra space allocated when OOB access is detected. These methods are intuitively straightforward and compatible to existing applications and libraries. Recent evaluations by [98] conclude that CRED is with the best detection response due to its high detection and low false alarm rate among seven well-known detectors. Their major drawback is a unacceptable high execution overhead, about six to thirty times slower.

Our system treats the full virtual address as a natural fence for OOB protection. Each allocated buffer will be randomly distributed among the virtual space with maximum distance separation from other buffers. Since most of the virtual space is within unallocated address range, any overflow, underflow, and hopping access of OOB will trigger page faults. These OOB pages are memory-mapped memory space with unallocated addresses. They virtually exist without occupying physical space. If the accesses are within bounds, the executions are the same as the original system without protection. If the accesses are out of bounds, the unallocated addresses will trigger our pre-specified signal handler.

Our key idea is simple but still completely preserves the capability of OOB detection and recovery based on J&K series of methods. The difficulties of implementation lie in the subtle use of mmap system call, allowing memory allocation within unallocated regions, and the scheme of faulty address resolution. It is a miniature memory management system and treats each buffer as a guarded memory region. Unlike the pure isolation purpose of [88]and similar to the page fault handler in the operating system kernel, our method requires to resolve faulty addresses in order to deal with the recovery process (discussed in Section 2.3).

5.2.2 Guarding Buffers with Addresses within Unallocated Regions

Current operating systems provide each process a virtual address space that is isolated from other processes. In the virtual address space, the stack resides at the top of the user process address space and heap resides behind the text and static data (including the initialized and uninitialized data). The region between the heap and the stack is an unallocated region, where normally no memory block resides at. The heap grows to higher addresses by invoking the `brk` or `sbrk` system call, while the stack grows to lower addresses as performed by the kernel automatically. Growing of the heap and stack is contiguous. Namely, a new allocated memory block is adjacent to the last allocated block. Note that growing the heap and stack reduces the size of the unallocated region.

Except growing the heap and stack, the `mmap` system call [72][41] can be used to allocate memory blocks within the unallocated region. The common usage to `mmap` is mapping files or devices into memory and therefore the mapped files or devices can be accessed in the same manner as accessing memory. This technique can be used to construct shared memory for an inter-process communication mechanism.

On the other hand, `mmap` can also be used to allocate memory blocks that are not associated with any file or device. Considering the behavior of allocation memory, the major difference between `mmap` and `brk/sbrk` is that `mmap` can specify the starting address of the allocated blocks, while `brk/sbrk` can not. That is, a new allocated memory block with `mmap` can be apart from the last allocated block. Therefore, `mmap` can allocate memory blocks discretely while `brk/sbrk` only allocates memory blocks continuously.

In order to efficiently detect and tolerate buffer overflows, the BODAR system utilizes the discrete allocation characteristic of `mmap` to introduce unallocated pages between allocated buffers. Note that an access to an unallocated page is an invalid memory reference

and triggers a segment violation signal (`SIGSEGV`) in UNIX or UNIX-like operating systems. If a buffer is followed with unallocated pages, any overflowing access out of the buffer bound causes a `SIGSEGV`. By carefully handling the signal, the buffer overflow can be detected and eliminated.

The buffer organization in the BODAR system is showed in Figure 4.1. Each allocated buffer is appended with a spare region and a forbidden region. A spare region consists of one or more unallocated pages while the forbidden region consists of only one unallocated page. If the size of a requesting buffer is larger than n pages but less than n+1 pages (n is zero or a positive integer), we allocate n+1 pages to the buffer and align the end of the buffer with the end of the n+1 pages.

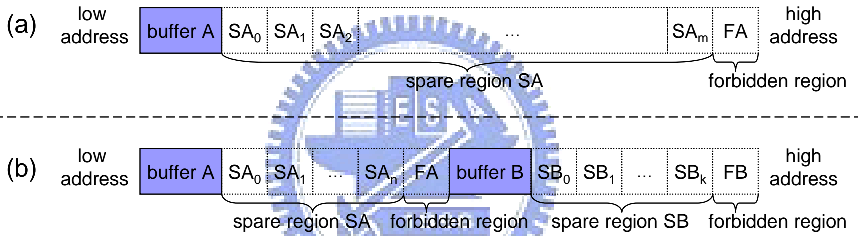


Figure 5.1. The buffer organization in the BODAR system.

A spare region that is designed to tolerate buffer overflows while the forbidden region is designed to set a limitation to the buffer overflow tolerance. In order to tolerate buffer overflows, it is better to make each spare region as large as possible. Hence the BODAR system allocates each buffer in the middle of the current largest spare region. For example, if the spare region `SA` is the largest one (shown in Figure 4.1(a)), the BODAR system positions the next allocated buffer B in the middle of `SA` (shown in Figure 4.1(b)). That is, n (the size of spare region `SA`) equals to k (the size of spare region `SB`) or n = k+1. Currently, we implement the allocation strategy with an AVL tree to maintain the buffer organization and a priority queue to track the size of each spare region so that the largest spare region can be found efficiently.

The BODAR system tolerates OOB accesses in the following ways:

- In Figure 5.1(a), assume that one unallocated page SA_x ($m \geq x \geq 0$) of spare region SA is read or written. This triggers the SIGSEGV signal and SA_x can be identified in the signal handler for SIGSEGV. The details of obtaining the faulty address SA_x is discussed in the next subsection.
- The BODAR allocates a page of memory at SA_x so that the program can continue to run without data corruption or termination. Hence the OOB reading before any writing at SA_x obtains a value zero.
- If the forbidden region FA is read or written, currently BODAR terminates the program to prevent the buffer behind A from modifications and therefore intrusions.
- The pages storing the OOB data in spare region SA are treated as an extension of buffer A. If buffer A is freed, they are freed too.
- To prevent OOB data from consuming too much memory, we set a limitation to the size of each buffer's extension, e.g., t pages ($t \geq 0$). If the unallocated page SA_x ($x>t$) is accessed, BODAR maps the device /dev/zero to the range from SA_t to SA_{x-1}. Since the pages mapped from /dev/zero are read-only, BODAR allocates a page of memory at SA_y ($x>y \geq t$) if the read-only page SA_y is written.

5.2.3 Faulty Address Resolution

In order to recover from an OOB access, we need to resolve the faulty address that causes the SIGSEGV signal. However, the widely-used ANSI C signal handler prototype (void signal_handler(int)) does not provide this information. Fortunately, the POSIX style signal handler prototype provides the information. Here, we describe how to use the POSIX style prototype to obtain the faulty address.

The POSIX style signal handler prototype:

```
    void signal_handler(int signo, siginfo_t *info, ucontext_t *uap);
```

The prototype listed above comes from FreeBSD. The faulty address is stored in the `si_addr` element of the second argument `info` that is a `siginfo_t` structure. To use this prototype, we must use the `sigaction` function to install a signal handler for `SIGSEGV` and turn on the `SA_SIGINFO` flag in the fourth argument `sa_flags` of `sigaction`.

This faulty address resolution method has been tested on Linux and FreeBSD. The third argument of the POSIX style prototype on Linux slightly differs from the FreeBSD one listed above but the difference does not affect the faulty address resolution.

5.2.4 Source Code Transformation for the Guarded OOB Instrument

We have proposed a discrete buffer allocation technique to recover from buffer overflows. In this subsection we discuss how to apply the proposed technique to heap buffers, and stack buffers respectively.

The proposed technique is easy to be applied to heap buffers since most ones are allocated with the `malloc` family and we can re-implement the `malloc` family based on the technique. By statically linking with applications or using the preloading dynamic loadable library technique [5], the new `malloc` family can replace the original one.

However, applying our technique to stack buffers is not as easy as applying to heap buffers. Stack buffers are allocated by merely moving the CPU frame pointer register. This process proceeds without any invocation of function calls. In order to apply the proposed technique to stack buffers, we decide to convert all stack buffers to heap buffers by rewriting the source code of applications without changing the semantics of programs. After such rewriting and recompiling, applications can utilize our technique to recover from stack buffer overflows.

We perform the code rewriting following the method described in [21]. The paper

proposes a tool named Gemini to convert all stack buffers to heap buffers in order to prevent stack buffer overflow attacks. It uses TXL [81] to perform the code rewriting. Note that TXL is a hybrid rule-based and functional language and is suitable for source code to source code rewriting. To perform the C code rewriting, we need the C grammar definition for TXL and a set of rule that specifies how to rewrite code based on the C grammar definition. The C grammar definition for TXL is available from the TXL website while the rule set was written by the authors. We also use Perl scripts to perform the code rewriting automatically.

Original code

```
void foo() {
  int buf[64];
  bzero(buf,sizeof(buf);
  ...
}
```

Modified code

```
void foo() {
  int *buf=(int*)malloc(64*sizeof(int));
  bzero(buf,64*sizeof(int));
  ...
  free(buf);
}
```

Figure 5.2. Converting stack buffers to heap buffers.

The example showed in the Figure 4.2 demonstrates the source code rewriting that converts stack buffers to heap buffers. In the original code, `buf` is a stack buffer in function `foo`. In the modified code, `buf` becomes a pointer and is initialized with a `malloc` that allocates a heap buffer with the same size of `buf` in the original code. Note that `buf` does not need to be reclaimed in the original code. This is because all stack buffers declared within a function are automatically reclaimed before the function returns. On the contrary, in the modified code `buf` must be explicitly reclaimed with the `free` function before `foo` returns since heap buffers are not reclaimed automatically.

The `sizeof` construct must be carefully handled. In the original code, `sizeof(buf)` represents the length of `buf`. However, in the modified code `sizeof(buf)` represents the length of a pointer since `buf` becomes a pointer after the rewriting. Our solution to this problem is replacing each affected `sizeof` construct with the constant that is the product of element count and the size of a single element.

5.3 Results

In this section we present the experiments to validate the proposed technique and evaluate the performance. We perform the validation and performance evaluation with several open source applications that have buffer overflow vulnerabilities. All hosts used in the experiments ran on FreeBSD 5.4 and each of them was equipped with 1GB RAM, 80GB hard disk, and a 100 Mb Ethernet card. One of them was 64-bit machine equipped with an AMD Athlon 64 3000+ CPU. Each of other hosts was 32-bit machine equipped with an AMD Athlon XP 2000+ CPU. They all connect to a 100 Mb switch hub directly without interference of irrelevant traffic. We conduct the evaluations in two major phases: one is to evaluate the feasibility and security tolerance of BODAR and the other is to perform efficiency evaluation. The results in the first phase are much satisfied for a better throughput performance than the original system without protection. The second phase results reveal the tradeoff between space and time. The execution efficiency is almost the same as the unpatched version with extra space overhead, on average ranging from 30% to six times larger.

5.3.1 Stack Buffer Allocation Overhead

The following experiment demonstrates the extreme case of buffer allocation overhead. It was performed and averaged by running the following program 50 millions times. The results exhibit the worst case of execution overhead due to space page allocation. However, in normal situations, buffer allocation usually constitutes only a small fraction of the overall runtime.

```
int test() {
  char buf[128];
  strcpy(buf, "1234567890");
  return strcmp(buf, "9876543210");
}
```

	Original	BODAR
Execution time (sec.)	2.0083	7.2168

Figure 5.3. The stack buffer allocation overhead.

5.3.2 Security Tolerance and Availability Evaluation

The evaluations of security tolerance are to prove the feasibility of our asynchronous OOB detector and recovering capability. It needs to preserve the original detection response (true negative and false positive rate) of CRED and BMB while retain better execution efficiency. We design the experiments on two kinds of representative servers: one is a multi-processes concurrent model using the Apache httpd and the other is a single-process concurrent model using the thttpd.

- **Process Pool Model Server**

The use of the Apache httpd is with the configuration of initial process pool size of 10 and the max process pool size of 150. We perform the evaluation on the http server running apache 1.3.34 with mod_mylo 2.1. The vulnerability does not reside in the main process of httpd but in the plugin of mod_mylo. We try to exploit the vulnerability of mod_mylo 2.1. In order to perform efficiency benchmark evaluation, we do not inject shell code from the exploit, but just crash the server. To use mod_mylo, the dynamic shared object (DSO) option of apache httpd must be enabled. The DSO option will slow down the httpd process. Both of our system BODAR and the compared target of CRED are with DSO enabled.

The following figures reveal the evaluation results by using the Webstone [47] web performance benchmarking system accompanying with the probe which exploits mod_mylo 2.1 vulnerability [67]. Three different versions of the Apache httpd are evaluated: the Original unpatched, CRED-patched, and our BODAR-patched httpd. The results are averaged for three runs of Webstone benchmarking and each time span of 3 minutes.

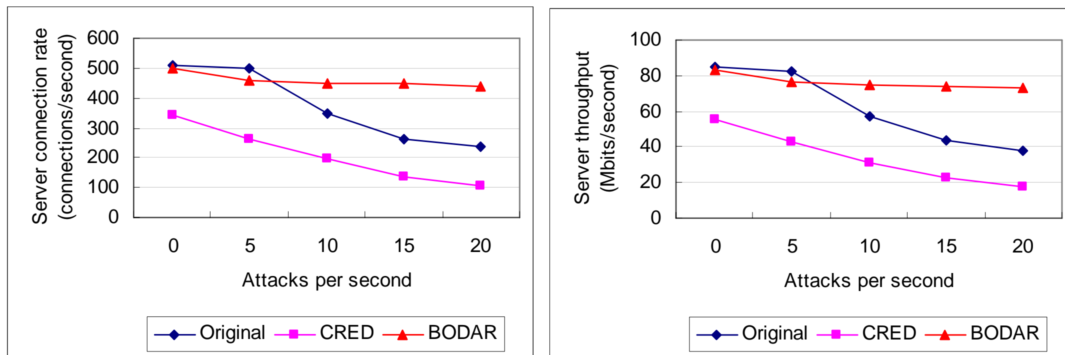


Figure 5.4. Apache http server 1.3.34 (with mod_mylo 2.1) runs under attacks (part 1).

The left shows the server connection rate and the right shows the server throughput. The connection rate of CRED-enabled server is about 33% slower while BODAR-enabled server is 30% faster than that of the original one. The improvement of throughput is even larger. The CRED server is 55% slower while the BODAR server is 92% faster than that of the original one.

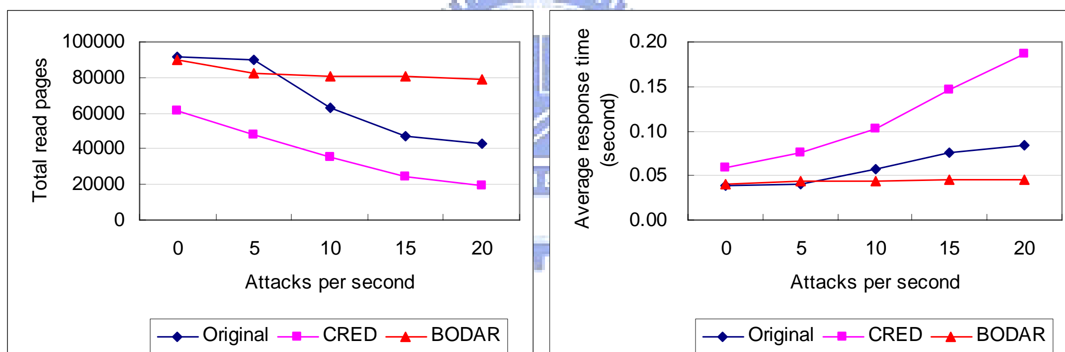


Figure 5.5. Apache http server 1.3.34 (with mod_mylo 2.1) runs under attacks (part 2).

The left shows the total number of pages read and the right shows the average response time. The total web pages read and average response time also reveal such improvement. The results indicate that the BODAR protection can be a good surviving technique for production purpose.

- **Event Driven Model Server**

The thttpd [1] is an event driven model server. The evaluation is also conducted by using Webstone with probe exploiting the thttpd vulnerability reported by [68].

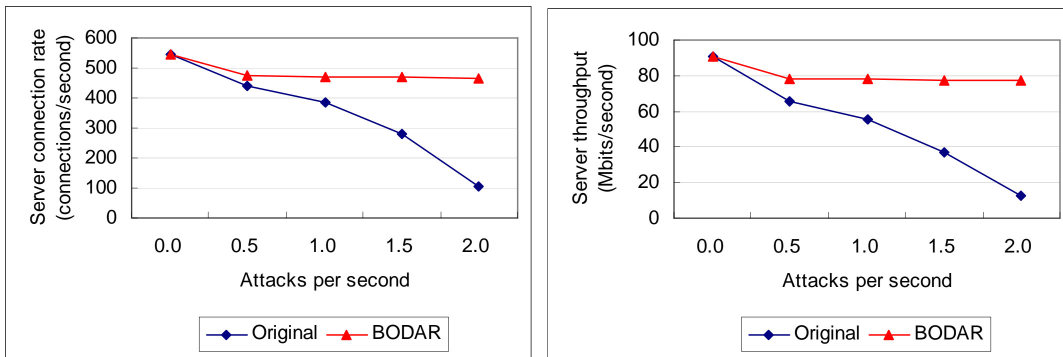


Figure 5.6. Thttpd 2.23beta1 runs under attacks (part 1).

The left shows the server connection rate and the right shows the server throughput. The connection rate of BODAR-enabled server is three times faster than that of the original one at the rate of 2 attacks/second. The improvement of throughput by the BODAR-enabled server is six times faster than that of the original one at the same attack rate.

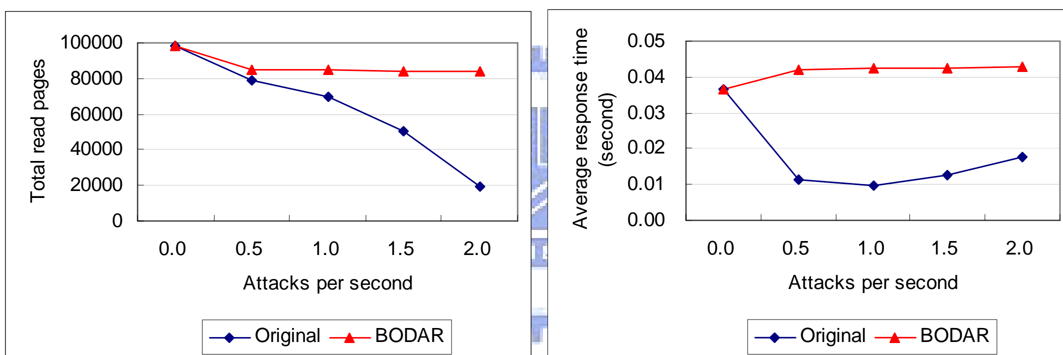


Figure 5.7. Thttpd 2.23beta1 runs under attacks (part 2).

The left shows the total number of pages read and the right shows the average response time. The total web pages read by the BODAR-enabled server are seven times larger. The response time of BODAR is longer than that of the original server is due to the handling of recovery while the original server just quickly responds with failures.

5.3.3 Efficiency Evaluation

To evaluate the normal execution efficiency without attacks, we classify the experiment benchmarks into I/O bound and CPU bound programs.

- **I/O Bound Programs**

We also compare three different versions of the Apache httpd and two versions of thttpd by the Webstone benchmarking system. The results are averaged in three runs with each time span of three minutes.

	apache			thttpd	
	Original	CRED	BODAR	Original	BODAR
Server connection rate (connections/sec.)	558.31	377.78	521.26	547.20	546.50
Server throughput (Mbits/sec.)	92.24	63.20	86.58	90.51	90.52
Total number of pages read (pages)	100496.33	67999.33	93826.33	98496.00	98370.67
Average response time (sec.)	0.035778	0.052848	0.038323	0.036508	0.036555
Maximum resident set size (Kbytes)	2545.33	N.A.	3134.08	8852.00	11692.00
Average shared text size (Kbytes)	359.23	N.A.	392.33	71.00	82.00
Data size (Kbytes)	2117.32	N.A.	2741.76	8781.00	11610.00

Figure 5.8. Apache http server 1.3.34 and thttpd 2.23beta1 runs without attacks.

Due to the complicated behavior nature of the process pool model, the space overhead is measured by the instrument of `malloc` family functions. By the registration of a function using `atexit`, we can obtain the residential memory usage of the current process with `getrusage` and record it by `syslog`. The CRED-enabled version will link the CRED provided `malloc` family functions and our space measurement won't work for CRED. However, CRED space overhead is not of our concerns.

	thttpd	apache
Maximum spare size (pages)	1842	7373
Average spare size (pages)	1239	5810
Minimum spare size (pages)	918	3683
Sparse degree	0.1787%	0.0239%

Figure 5.9. Sparse Degree Measurement.

We define sparse degree to be (sum of all buffer size) / (utilizable virtual space size). The more sparse the overall virtual space is used, the smaller possibility the hopping can override existing buffers.

- **CPU Bound Programs**

We choose the benchmark for digital signing purpose with gnupg 1.4.1. Since the primary function of signing application is for the computation of the message digest and the

signature, the execution time is in proportion to the size of the data file. We exploit the vulnerability of gnupg and both of the CRED and BODAR can detect the OOB access.

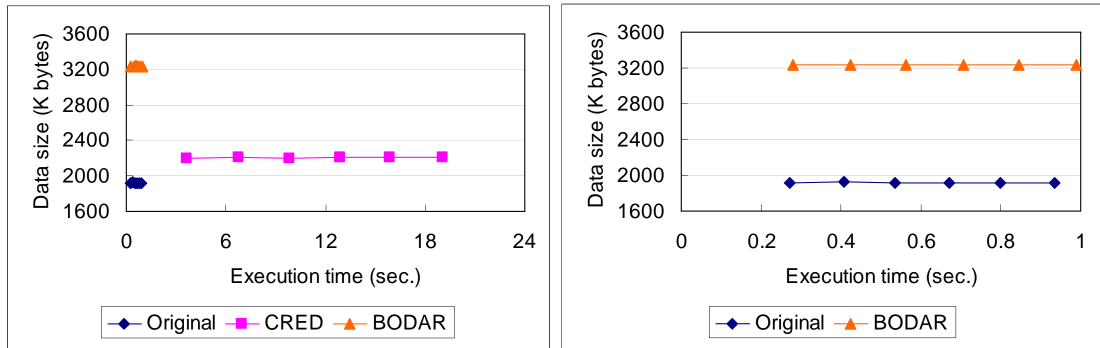


Figure 5.10. Use GnuPG to perform RSA encryption. The source file sizes are 0.5M, 1M, 1.5M, 2M, 2.5M, and 3M bytes files respectively.

By the experiment of RSA encryption of GnuPG, the above results also reveal that the CRED performs about twenty times slower than the original version. The BODAR system is about 5% slower at the extra space overhead of 25% larger.

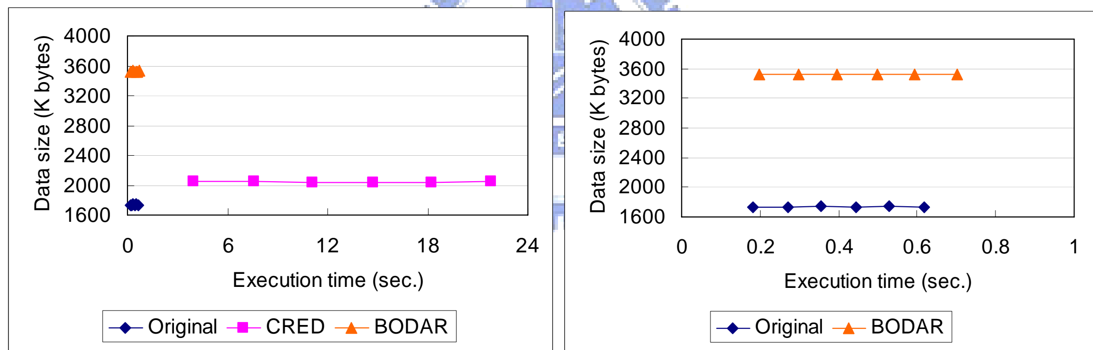


Figure 5.11. Use GnuPG to perform RSA decryption. The decrypted file sizes are 0.5M, 1M, 1.5M, 2M, 2.5M, and 3M bytes respectively.

By using the GnuPG to perform RSA decryption, the CRED is thirty-five times slower while the BODAR is only 7% slower. The extra space overhead is about 43% larger.

To measure our worst cases of space overhead, we perform the benchmark evaluation by gawk 3.1.0 with overflow vulnerability.

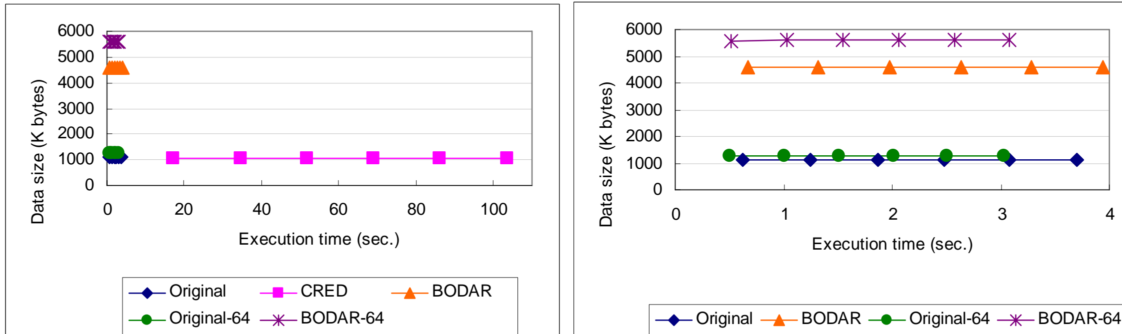


Figure 5.12.Use access-time.awk to analyze 50K-lines, 100K-lines, 150K-lines, 200K-lines, 250K-lines, and 300K-lines squid access log respectively.

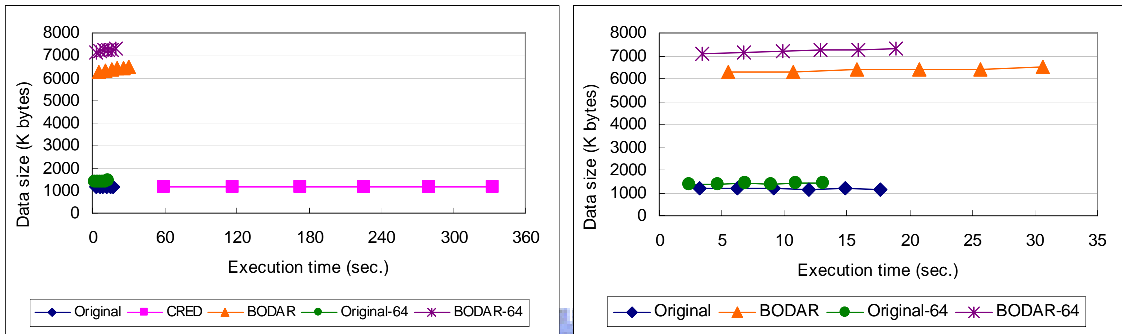


Figure 5.13. Use proxy_stats.awk to analyze 50K-line, 100K-lines, 150K-lines, 200K-lines, 250K-lines, and 300K-lines squid access log respectively.

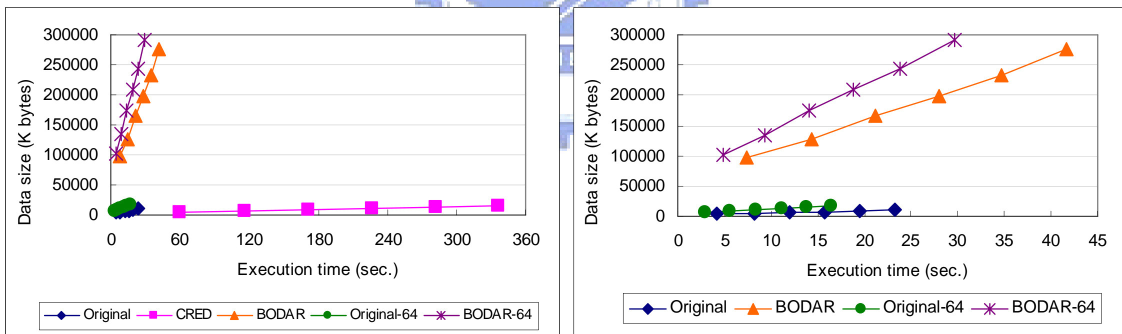


Figure 5.14. Use scalar.awk to analyze 50K-lines, 100K-lines, 150K-lines, 200K-lines, 250K-lines, and 300K-lines squid access log respectively.

The above results are obtained with three awk benchmark scripts: access-time.awk, proxy_stats.gawk and scalar.awk for analysis of log files with size of 50K, 100K, 150K, 200K, 250K, and 300K lines respectively. The CRED version is about 10 times to 30 times slower while the BODAR is about 20% slower. The space overhead of BODAR is 2.5 times to 28 times larger. This exhibits the worst situations of space consumption of the BODAR tradeoff between time and space.

	gpg		gawk (32-bit)			gawk (64-bit)		
	encrypt	decrypt	script 1	script 2	script 3	script 1	script 2	script 3
Maximum spare size (pages)	3684	3722	1841	919	26	134180344	69825501	1708544
Average spare size (pages)	3220	2443	1083	586	17	79674386	43001310	1182793
Minimum spare size (pages)	1834	1836	918	458	11	67090170	33545083	853750
Sparse degree	0.0453%	0.0459%	0.0937%	0.1827%	14.2686%	0.0000013%	0.0000026%	0.0002022%

Figure 5.15. Sparse Degree of CPU bound programs.

The above results are measured when the maximum number of pages is allocated. The gnupg case is with a three megabytes file and gawk is with 300 K-lines of Squid access log.

5.4 Discussions

5.4.1 The Utilizable Virtual Address Space

Since the proposed technique requires extra virtual address space that is limited for each process, we need to find out the utilizable virtual address space.

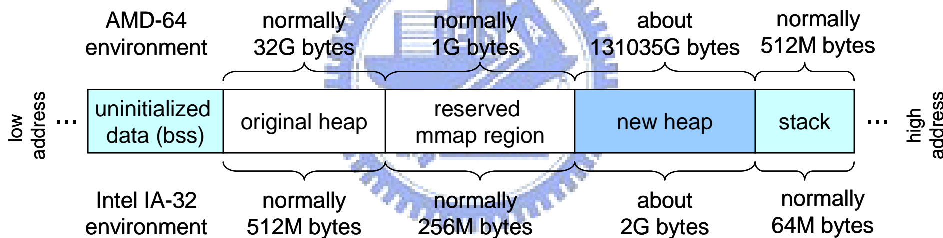


Figure 5.16 The memory layout of a BODAR-enabled process on FreeBSD 5.4 on an AMD-64 machine or an Intel IA-32 machine.

Figure 5.16 shows the memory layout of a BODAR-enabled process on FreeBSD 5.4 on an AMD-64 machine or an Intel IA-32 machine. The region used by BODAR is called new heap. The region behind the new heap region is stack. The region ahead of the new heap region is called the reserved `mmap` region that is reserved for compatibility since applications may use no-starting-address-hint `mmap` calls (`mmap` calls without specifying the starting address of the requesting pages) to map something, e.g. files, into their address spaces. We set the size of reserved `mmap` region to be 1G bytes and 256M bytes on an AMD-64 machine and an IA-32 machine respectively.

The region including the reserved mmap region and the new heap region can be called the mmap controllable region since both them are controlled by mmap. The size of new heap region can be determined by subtracting the size of reserved mmap region from the size of mmap controllable region.

The lower bound of the mmap controllable region can be determined by the first no-starting-address-hint mmap call since a process begins. On the other hand, the upper bound of the mmap controllable region can be determined by the following steps.

1. Determine the upper bound of stack. Since the location of environment variables is near the upper bound of stack, we approximate the upper bound of stack to the address that is gotten by rounding up the address of the last environment variable to the page boundary.
2. Subtract the maximum size of stack from the upper bound of stack. The maximum size of stack can be obtained by the getrlimit function.
3. Since stack is behind the mmap controllable region, we set the upper bound of the mmap controllable region to be the lower bound of stack.

By subtracting the lower bound from the upper bound of the mmap controllable region, the size of mmap controllable region and therefore the size of new heap region can be determined. On FreeBSD 5.4, the size of new heap region is about 131035G bytes on an AMD-64 machine and 2G on an Intel IA-32 machine. We wrote a simple program to allocate each page in the new heap region and have proven that each page in the new heap region is able to be allocated by mmap on both AMD-64 and Intel IA-32 machines on FreeBSD 5.4.

Since each allocation of BODAR requires at lease 3 pages of virtual address space, in worst case the maximum heap size of a BODAR-enabled process is about 680M bytes (2.0 G bytes dividing 3) on an IA-32 environment. We believe that this heap size is enough for many applications. On an AMD-64 environment, the maximum heap size is about 43678G bytes

(131035G bytes dividing 3).

5.4.2 Limitations

Our design of asynchronous OOB detection and recovery is still with some limitations. We summary them as follows:

1. Buffers within a struct or class are not guarded. We haven't dealt with the buffer embedded within the class or struct fields. It can be further processed by a sophisticated parser.
2. The BODAR design does not deal with static buffers due to the following two reasons: first, most memory accesses occur in heap and stack; second, most reported buffer overflow vulnerabilities also occur in heap and stack buffers. Currently we do not deal with static buffers for the sake of simplicity, but we will study how to apply our technique to static buffer in the future.

Chapter 6 Conclusion

In this dissertation, we present the GOTG system for online tabletop games and two related issues on which the services of our system are blocked or disrupted. We summarize these research results and discuss some future work in this chapter.

6.1 Summary of Contributions

In this dissertation, we design and implement the GOTG system in order to facilitate the development of online tabletop games. We first survey the similarity of online tabletop games and summarize the shared characteristics. According to the shared characteristics, we design two models. The first one is the GOTG model that allows players to search for their friends in a virtual room and then play games with them in the room. The second one is the TG model that defines game objects, game states, and playing operations for tabletop games. For the TG model, we formalize the definition of tabletop games and related terminologies and then prove that it is general for all tabletop games.

Based on the two models, we implement the GOTG system that allows players to search for their friend and play tabletop games over the Internet. For developing an online tabletop game, the GOTG system provides the supports for network communication, player management, matchmaking, etc. Besides, the GOTG system includes a framework based on the TG model that provides graphical supports for tabletop games. The framework helps developers handling game areas and game objects used in tabletop games. It supports move-object, max-Z, and next-face operations for game objects.

The GOTG system provides services for playing tabletop games over the Internet. While implementing the GOTG system practically, we encounter two issues on which the services of our system for online tabletop games are blocked or disrupted. In order to provide the services

correctly and smoothly, we try to solve the two issues. We present our solutions to them in chapter 4 and 5 respectively and summarize the results as follows.

- Resolving problems of blocking I/O operations for server programming

  Event-driven programming is a widely used technology for concurrent programming. However, the major drawback of event-driven programming is the need to pay attention to the blocking I/O operations in event handling. This dissertation addressed two major blocking problems due to the use of blocking I/O operations in game servers and other inter-user communication applications (defined in Section 4.1), namely output blocking and service request blocking. For the former, this dissertation presents an output buffering mechanism to solve this problem. Meanwhile, for the latter, this dissertation presents a service brokering mechanism with helper processes to solve this problem.

  Based on the output buffering and service brokering mechanisms, we design an event-driven framework for inter-user communication applications, as shown in Figures 4.5 and 4.9. Application developers can apply this framework to develop their servers simply by implementing some event handlers and service handlers, as in the shadowed classes in Figures 4.5 and 4.9. Therefore, based on this framework, they can easily avoid blocking problems. Besides, the framework has been the basis of the network module of the GOTG system so that the game servers in the GOTG system are immune from the two blocking problems.

  In fact, our framework can also be applied to other applications. For example, event-driven based HTTP servers, such as Zeus [97], mathopd [11], and thttpd [1], can be implemented on top of this framework by putting file access operations into helper processes.

- Fast recovery from overflow failures for non-disruptive game services

  The services of the GOTG system rely on the collaboration of several kinds of servers.

Since we expect to provide the services without any disruption, the servers should try to recover from failures timely. One of failures that we encountered when developing the GOTG system is the buffer overflow failure that is commonly found in many software recently. In the dissertation, we develop a technique named Buffer Overflow Detection and Recovery (BODAR) to detect and recover from buffer overflow failures efficiently. The design of BODAR is a miniature of user space management in the kernel of operating system. Each allocated buffer is considered as isolated and guarded regions. With the allocation of guarded regions and faulty address resolution scheme, we can transform all J&K series of OOB detectors and recovery tools into asynchronous execution. Therefore, in normal situations without attacks or OOB accesses, the execution efficiency of BODAR maintains about the same as those without protection, while achieving the same security tolerance of the above OOB detectors at the expense of extra space overhead. The impact of BODAR resides in security considerations in comparison with Stackguard series and J&K series of OOB detectors. Stackguard series of OOB detectors simply terminate the victim processes and still suffer from denial of service problems since large number of malicious connections will very likely incur frequent service shutdowns. BODAR can remedy this denial-of-service problem by efficiently recovering from malicious connections as evidenced by experiments on Webstone benchmark with OOB exploits. The experimental results show better throughput than the original server. In regard of J&K series of OOB detectors, execution efficiency is a major concern. These detectors can only be used for the development phase instead of the production purpose. However, since the absence of OOB bugs is undecidable, an OOB detector for production use is essential. Our BODAR design can resolve such security concerns.

Since the GOTG system provides the supports for network communication, player management, matchmaking, game object and operation handling, etc., it can effectively

reduce the effort to develop online tabletop games. With the help of the GOTG system, developers can focus on the design and implementation of the game rule.

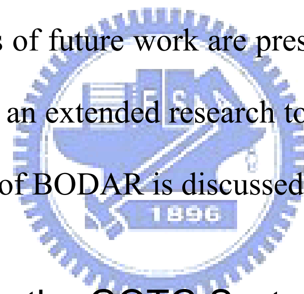
Practically, the GOTG system has been used in the CYC Game League [82], Sina [70, 71], Hinet [16], and other game sites in Taiwan and Hong-Kong. These sites provide dozens of tabletop games, such as Chinese Chess, Bridge, Mahjong, etc. Currently, these sites totally have attracted more than one million people to register as members and supported up to 10,000 concurrent players. Figure 6.1 and 6.2 are the screenshots of the customized GOTG system in the CYC Game League. Figure 6.1 is the screenshot of a game coordinator in which 52 players login to play Chinese Chess. Figure 6.2 is the screenshot of a Chinese Chess game in which two players are playing.

Figure 6.1. The screenshot of a game coordinator in the CYC Game League.

Figure 6.2. The screenshot of a Chinese Chess game in the CYC Game League.

6.2 Future work

In this section, several pieces of future work are presented. First, an improvement on the GOTG system is discussed. Then, an extended research to the TG model is presented. Finally, a method to reduces the overhead of BODAR is discussed.



6.2.1 The Improvements on the GOTG System

The improvement on the GOTG system is to share the network traffic of game servers to clients. Currently, a game server in the GOTG system is responsible to relay the actions or messages of one player to others in a game session. For example, in a GOTG-based Bridge game, the action of one player dealing out a card is firstly sent to the game server. Then the server relays the action to the other three players. As a result, the server may send at least three times of data than it received in many cases. So that the game server may become the bottleneck in data transmission. We plan to use the peer-to-peer (P2P) technique to resolve this problem. That is, we wish to design a method to find some clients to be the super nodes that help the game servers to relay game actions and messages. This mechanism should be integrated into the network module of the GOTG system and be transparent for developers.

6.2.2 GAML and History Authoring Systems

Chess-like games, such as Chess, Chinese Chess, Go, and Gobang, are popular tabletop games in the world. Due to the popularity, several file formats are defined for storing game records in order to help players learning Chess-like games. However, these formats are usually dedicated for a small set of games. For example, the well-known Smart Game Format (SGF) [32] is dedicated for Go and several games. Besides, there may be several file formats for a Chess-like game. For example, there are at least four formats [4] for Chinese Chess. When Chess-like games need to exchange, many formats make the interoperability hard.

Since the TG model is general for tabletop games, the we are motivated to design a general file format called GAML for storing game records. GAML is based on XML [86] in order to take the advantages from XML. It can be an intermediate format to make the translation among existing formats easier. Furthermore, a history authoring framework based on GAML for Chess-like games may be defined so that it becomes efficient to develop a history authoring system for each game based on the framework.

6.2.3 The Improvement on BODAR

The major problem of our BODAR system is the space overhead. In normal situations, extra space overhead is on average around 25%. In worst situations, it can consume as much as thirty times extra space. In the future, we shall further improve BODAR to reduce the space consumption. Since the space overhead of BODAR is introduced from the internal fragmentation of spare and forbidden pages, we propose to adaptively adjust BODAR's use of protected buffers. A possible adjustment is to change the protection of small and rarely used buffers by using CRED or address obfuscations [7][5] and protection of the remaining large or frequently used buffers by BODAR. Using this adaptive scheme for security protection and tolerance, we aim to optimize the tradeoff between time and space usage. If space is more

concerned than time, BODAR zone can be reduced according to the past working model of small and rarely used buffers. Otherwise, BODAR zone will override and guard all allocated buffers.

To apply the BODAR protection in more ambitious ways, we can guard frequently used and automatic expanded arrays in various script languages such as Perl, PHP, and Python, and also boundary checking arrays in the Java's `StringBuffer` objects. After the feasible trial of miniature for the isolated buffer in a dedicated process space, all of the above protections can then be realized in the BODAR approach.

References

[1] ACME Laboratories. thttpd. http://www.acme.com/software/thttpd/ (last access: May 2005).

[2] T. Austin, S. Breach and G. Sohi. "Efficient detection of all pointer and array access errors." In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 2004.

[3] I. Baldine. Divert Sockets mini-HOWTO. http://www.faqs.org/docs/Linux-mini/Divert-Sockets-mini-HOWTO.html (last access: April 2006).

[4] S. Bao. XQFtoAny. http://www.hgchess.com/ReadNews.asp?NewsID=355 (last access: May 2006).

[5] A. Baratloo, N. Singh and T. Tsai. "Transparent Run-Time Defense against Stack-Smashing Attacks." In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pp. 251, June 2000.

[6] Beyond Security. SecuriTeam. http://www.securiteam.com/ (last access: May 2006).

[7] S. Bhatkar, D.C. DuVarney and R. Sekar. "Address Obfuscation: An Efficient Approach to Combat A Broad Range of Memory Error Exploits." In *USENIX Security Symposium*, pp. 105-120, August 2003.

[8] BioWare. Baldur's Gate. http://www.bioware.com/games/baldurs_gate/ (last access: May 2006).

[9] Blizzard Entertainment Inc. Blizzard Entertainment – Warcraft III. http://www.blizzard.com/war3/ (last access: April 2006).

[10] Blizzard Entertainment Inc. Diablo II. http://www.blizzard.com/diablo2/ (last access: May 2006).

[11] M. Boland. Mathopd. http://www.mathopd.org/ (last access: April 2006).

[12] G. Booch, I. Jacobson and J. Rumbaugh. *The Unified Modeling Language User Guide.* Addison-Wesley, Massachusetts, 1998.

[13] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz and K.J. Worrell. "A Hierarchical Internet Object Cache." In *Proceedings of the 1996 USENIX Technical Conference*, San Diego, California, USA, pp. 153-163, 1996.

[14] M. Chew and D. Song. "Mitigating buffer overflows by operating system randomization." Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[15] T.C. Chiueh and F.H. Hsu. "Rad: A compile-time solution to buffer overflow attacks." In *21st International Conference on Distributed Computing*, Phoenix, Arizona, USA, pp. 409, 2001.

[16] Chunghwa Telecom. HiNet Game Zone. http://games.hinet.net/ (last access: May 2006).

[17] A. Cobbs. Divert. http://www.freebsd.org/cgi/man.cgi?query=divert (last access: April 2006).

[18] D.E. Comer. *Internetworking with TCP/IP Vol. 1: Principles, Protocols, and Architecture, Forth Edition.* Prentice Hall, New Jersey, 2000.

[19] D.E. Comer and D.L. Stevens. *Internetworking with TCP/IP Vol. 3: Client-Server Programming and Applications – BSD Socket Version, Second Edition.* Prentice Hall, New Jersey, 1996.

[20] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang. "StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks." In *Proceedings of the 7th USENIX Security Symposium*, pp. 63-78, January 1998.

[21] C. Dahn and S. Mancoridis. "Using Program Transformation to Secure C Programs Against Buffer Overflows." In *IEEE Proceedings of the 2003 Working Conference in Reverse Engineering (WCRE'03)*, British Columbia, Canada, pp. 323-332, November 2003.

[22] Electric Arts Inc. ORIGIN – Ultima Online. http://www.uo.com/ (last access: April 2006).

[23] H. Etoh and K. Yoda. Protecting from Stack-Smashing Attacks. http://www.trl.ibm.com/projects/security/ssp/main.html (last access: January 2006).

[24] FICS. Free Internet Chess Server. http://www.freechess.org/ (last access: April 2006).

[25] Firaxis Games. Civilization IV. http://www.2kgames.com/civ4/home.htm (last access: May 2006).

[26] M. Fisher, J. Ellis and J. Bruce. *JDBC API Tutorial and Reference, Third Edition.* Addison-Wesley, New Jersey, 2003.

[27] S. Forrest, A. Somayaji and D.H. Ackley. "Building diverse computer systems." In *6th Workshop on Hot Topics in Operating Systems*, Los Alamitos, CA, USA, pp. 67-72, 1997.

[28] J. Foster, M. Fahndrich and A. Aiken. "A theory of type qualifiers." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[29] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 1995.

[30] GigaMedia. FunTown. http://www.funtown.com.tw/ (last access: May 2006).

[31] R. Hastings and B. Joyce. "Purify: A tool for detecting memory leaks and access errors in C and C++." In *Proceedings of the Winter USENIX Conference*, pp. 125-138, January 1992.

[32] A. Hollosi. SGF File Format. http://www.red-bean.com/sgf/ (last access: May 2006).

[33] IBM. DeepBlue. http://www.research.ibm.com/deepblue/ (last access: May 2006).

[34] id Software. DOOM 3. http://www.doom3.com/ (last access: May 2006).

[35] Institute for Information Industry. View Point of MIC. http://mic.iii.org.tw/pop/micnews2_op_new.asp?sno=110&cred=2005/7/8 (last access: May

2006).

[36] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney and Y. Wang. "Cyclone: Asafe dialect of C." In *Proceedings of the USENIX Annual Technical Conference*, pp. 275-288, June 2002.

[37] R. Jones and P. Kelly. "Backwards-compatible bounds checking for arrays and pointers in C programs." In *Third International Workshop on Automated Debugging*, May 1997.

[38] S.C. Kendall. "Bcc: run-time checking for c programs." In *Proceedings of the USENIX Summer Conference*, 1983.

[39] A. Kirmse, "A Network Protocol for Online Games," in *Game Programming Gems,* M.A. DeLoura Ed. Massachusetts: Charles River Media, INC., 2000, pp. 104-108.

[40] D. Larochelle and D. Evans. "Statically Detecting Likely Buffer Overflow Vulnerabilities." In *Proceedings of the 10th USENIX Security Symposium*, pp. 177-190, August 2001.

[41] M.K. McKusick, K. Bostic, M.J. Karels and J.S. Quarterman. *The Design and Implementation of the 4.4BSD Operation System.* Addison-Wesley, Massachusetts, 1996.

[42] Microsoft. Age of Empires. http://www.microsoft.com/games/empires/ (last access: May 2006)

[43] Microsoft Corporation. Active Server Pages. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/activeservpages.asp (last access: April 2006).

[44] Microsoft Corporation. Internet Information Services. http://www.microsoft.com/WindowsServer2003/iis/default.mspx (last access: April 2006).

[45] Microsoft Corporation. MSDN: ODBC. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/HTML/_core_odbc.asp (last access: April 2006).

[46] Microsoft Corporation. MSDN: Thread Stack Size. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/thread_stack_size.asp (last access: April 2006).

[47] Mindcraft Inc. Webstone. http://www.mindcraft.com/benchmarks/webstone/ (last access: January 2006).

[48] Mudge. How to write buffer overflows. http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html (last access: May 2006).

[49] Multimedia University. Multiplayer Board Game Framework. http://foe.mmu.edu.my/software/gameframework/ (last access: May 2006).

[50] G.C. Necula, S. McPeak and W. Weimer. "CCured: Type-Safe Retrofitting of Legacy Code." In *Proceedings of the Principles of Programming Languages (PoPL)*, pp. 128-139, January 2002.

[51] B. Nichols, D. Buttlar and J.P. Farrel. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly, California, 1996.

[52] H. Nielsen, T. Berners-Lee and J. Groff. Libwww – the W3C Sample Code Library. http://www.w3.org/Library/ (last access: April 2006).

[53] NKB Inc. Internet Go Service. http://igs.joyjoy.net/ (last access: April 2006).

[54] OpenLDAP Foundation. OpenLDAP Software Man Pages: ldap. http://www.openldap.org/software/man.cgi?query=ldap (last access: April 2006).

[55] J. Orwant, "Eggg: Automated programming for game generation," *IBM Systems Journal,* vol. 39(3):782–794, 2000.

[56] J. Orwant. "EGGG: The Extensible Graphical Game Generator." *PhD Thesis, MIT,* December 1999.

[57] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). http://home.pacbell.net/ouster/threads.pdf (last access: April 2006).

[58] V. Pai, P. Druschel and W. Zwaenepoel. "Flash: An Efficient and Portable Web Server." In *Proceedings of USENIX Annual Technical Conference*, Monterey, California, USA, pp. 199-212, 1999.

[59] B. Perens. "Electric Fence Malloc Debugger." In *Pixar Animation Studios*, 1993.

[60] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, Massachusetts, 1995.

[61] E. Rescorla and A. Schiffman. "The Secure HyperText Transfer Protocol." Technical Report RFC 2660, August 1999.

[62] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy and T. Leu. "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)." In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, 2004.

[63] R. Rugina and M. Rinard. "Symbolic bounds analysis of pointers, array indices, andaccessed memory regions." In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pp. 182-195, June 2000.

[64] O. Ruwase and M.S. Lam. "A Practical Dynamic Buffer Overflow Detector." In *Proceedings of the 11th Annual Network & Distributed System Security Symposium*, pp. 159-169, February 2004.

[65] D.C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design,* Massachusetts: Addison-Wesley, 1995, pp. 529-545.

[66] D.C. Schmidt, M. Stal, H. Rohnert and F. Buschmann. *Pattern-Oriented Software Architecture Vol. 2: Patterns for Concurrent and Networked Object.* John Wiley & Sons, New York, 2000.

[67] SecurityFocus. Mod_mylo apache module REQSTR buffer overflow vulnerability. http://www.securityfocus.com/bid/8287/ (last access: January 2006).

[68] SecurityFocus. Thttpd defang remote buffer overflow vulnerability. http://www.securityfocus.com/bid/8906/ (last access: January 2006).

[69] SEGA and Creative Assembly. Total War. http://www.totalwar.com/ (last access: May 2006).

[70] Sina. SinaHK. http://game.sina.com.hk/cgi-bin/index.cgi (last access: May 2006).

[71] Sina. SinaTW. http://games.sina.com.tw/ (last access: May 2006).

[72] W.R. Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Massachusetts, 1992.

[73] W.R. Stevens. *TCP/IP Illustrated Vol. 1: The Protocols.* Addison-Wesley, Massachusetts, 1994.

[74] W.R. Stevens. *UNIX Network Programming Vol. 1: Networking API: Sockets and XTI, Second Edition.* Prentice Hall, New Jersey, 1998.

[75] Sun Microsystems Inc. JavaServer Pages Technology. http://java.sun.com/products/jsp/ (last access: April 2006).

[76] Sun Microsystems Inc. JDBC Technology. http://java.sun.com/products/jdbc/ (last access: April 2006).

[77] Tanguy Urvoy and gnugo team. Pattern Matching in Go with DFA. http://www.irisa.fr/galion/turvoy/papers/dfabstract.ps (last access: May 2006).

[78] The Apache Software Foundation. The Apache HTTP Server Project. http://httpd.apache.org/ (last access: May 2006).

[79] The MITRE Corporation. Common Vulnerabilities and Exposures. http://cve.mitre.org/ (last access: May 2006).

[80] The PHP Group. PHP: Hypertext Preprocessor. http://www.php.net/ (last access: April 2006).

[81] The Software Technology Laboratory, Queen's University, Kingston, Canada. TXL Home Page. http://www.txl.ca/ (last access: January 2006).

[82] ThinkNewIdea Inc. CYC Game League. http://cycgame.com (last access: April 2006).

[83] J. Tranter, "Exploring the sendfile system call," *Linux Gazette,* vol. Issue 91, June. 2003.

[84] Valve. Half Life 2. http://half-life2.com/ (last access: May 2006).

[85] Vendicator. Stack shield. http://www.angelfire.com/sk/stackshield/ (last access: January 2006).

[86] W3C. Extensible Markup Language (XML). http://www.w3.org/XML/ (last access: May 2006)

[87] D. Wagner, J.S. Foster, E.A. Brewer and A. Aiken. "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities." In *Network and Distributed System Security Symposium*, pp.

3-17, February 2000.

[88] R. Wahbe, S. Lucco, T.E. Anderson and S.L. Graham. "Efficient software-based fault isolation." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, United States, pp. 203-216, December 05-08.

[89] M. Welsh, D. Culler and E. Brewer. "SEDA: An Architecture for Well-Conditioned Scalable Internet Services." In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Alberta, Canada, pp. 230-243, 2001.

[90] D. Wessels . Squid Web Proxy Cache. http://www.squid-cache.org/ (last access: May 2005).

[91] R. Wu and D.F. Beal, "Solving Chinese Chess Endgames by Database Construction," *Information Sciences,* 2001.

[92] I.C. Wu and D.Y. Huang. "A New Family of k-in-a-row Games." In *The 11th Advances in Computer Games (ACG11) Conference*, Taipei, Taiwan, 2006.

[93] I.C. Wu, D.Y. Huang and H.C. Chang, "Connect6," *Internation Computer Game Association,* 2005.

[94] J. Xu, Z. Kalbarczyk and R.K. Iyer. "Transparent runtime randomization for security." Technical Report UIUL-ENG-03-2207, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, May 2003.

[95] Yahoo! Inc. Yahoo! Games. http://games.yahoo.com/ (last access: April 2006).

[96] S.J. Yen, J.C. Chen, T.N. Yang and S.C. Hsu, "Computer Chinese Chess," *ICGA Journal,* March 2004.

[97] Zeus Technology limited. Zeus Web Server. http://www.zeus.co.uk/products/zws/ (last access: April 2006).

[98] M. Zhivich, T. Leek and R. Lippmann. "Dynamic Buffer Overflow Detection." In *2005 Workshop on the Evaluation of Software Defect Detection Tools (BugWorkshop05)*, Chicago, 12 June 2005.