Re-Usable Communications Software for Distributed Interactive Applications

B. Clement T. Kelly A. Nierman L. Opyrchal
{bradc,tpkelly,andrewdn,lukasz}@eecs.umich.edu
Artificial Intelligence Lab
University of Michigan
Ann Arbor, MI 48109 USA

December 16, 1997

Abstract

We describe an approach to communications infrastructures for networked applications which emphasizes layered, structured, and re-usable software. Our method is appropriate for applications in which human users and/or software agents at different hosts make frequent changes to replicated shared state variables, e.g., multi-player video games, shared whiteboard systems, and chat programs.

Many published discussions of networked games claim that a tight coupling between application-level semantics and network communications software is necessary for good performance. We briefly review a very simple system called "Dr. T" which presents a very clean and simple interface yet can support large numbers of players in a fast-paced game on LANs and campus-sized WANs. We then describe in detail a general-purpose communications system called "Mr. Hide" which exploits knowledge of application state to minimize communications overheads for applications with a spatial structure, e.g. games and shared whiteboards. Mr. Hide presents a sufficiently clean interface to be re-usable in a variety of applications.

This document is available in HTML on the World Wide Web at
`http://www-personal.engin.umich.edu/~tpkelly/ds2/`

`$Id: ds2.tex,v 1.9 1997/12/16 09:54:18 tpkelly Exp $`

1 Introduction

Several recent papers on the design of distributed multi-player games implicitly suggest that to achieve acceptable performance a game must be internally monolithic in the sense that networking protocols must be tightly coupled with the main game software [34, 15]. Recent work on a very simple general-purpose communications infrastructure for distributed interactive applications called "Dr. T" casts doubt on this idea. Dr. T presents a clean interface to application-level software yet can keep pace with a 30-Hz screen refresh rate in a game with over 30 players on a LAN. Measurements of network performance lead us to believe that a more sophisticated implementation of Dr. T might support over 50 players at 30 Hz on a building-sized WAN (one router), and over 15 players on a campus-sized WAN (several routers). For a detailed description of Dr. T, see [18].

Dr. T presents a very simple abstraction to application-level software, and does not meddle in the business of applications that use it. This paper describes a more complicated system called "Mr. Hide" that exploits some knowledge of application-level semantics in order to reduce communication overhead and thereby improve performance in distributed applications with a spatial nature, e.g. games and shared whiteboard systems. Although not as completely decoupled from application-level semantics as Dr. T, Mr. Hide is not as tightly integrated with application-level software as the communications schemes described in some literature on distributed games. Mr. Hide represents a compromise between the extremes of complete modularity and complete integration.

This paper is organized as follows. Section 2 describes the design of Mr. Hide. Section 3 describes the group communications system beneath Mr. Hide. Section 4 relates our experiences in using Dr. T and Mr. Hide with a "legacy code" application. Section 5 summarizes the lessons we learned from our work with distributed interactive applications. We assume that the reader is familiar with our preliminary design document for Dr. T and Mr. Hide [6].

2 Mr. Hide System Design

2.1 Design Decisions

The basic strategy of Mr. Hide is to exploit higher-level semantics of multi-player games and similar applications in order to provide an efficient communication infrastructure. Maintaining consistent state replicas among distributed sites is generally handled by providing a policy for reading and writing shared variables. By considering special characteristics common to multi-player games, we can rule out some models of consistency which are inappropriate. For a detailed description of the Mr. Hide approach to *lazy state updating* in distributed multi-player games, see our preliminary design document [6]. In what follows we assume that the reader is familiar with the basic ideas outlined in the design document.

2.1.1 Replicating State

Treating a game as a distributed-shared memory problem would mean that state variables are partitioned among players, and there would be a cache-consistency protocol for accessing shared memory. However, this would mean that any one player will need to communicate with all others to access shared memory even though they may not interact with each other in the game environment for some time. We do not want to repeatedly update the cached state of a player that does not need access to the variables being updated. Another approach is to let one process manage shared memory, but that would result in a communication bottleneck at the central manager, hindering scalability.

Instead, it is better to limit players to communicate only to those with whom they are interacting. Access to shared variables can be limited to the players which could possibly need them. By taking advantage of information about the spatial environment, we can divide shared variables among the players that are in the region where the variables are available. Players can have replicated states that are only updated by players within their area of interaction. The difficulty in this approach is that players can move throughout regions of the environment and may need access to different sets of variables at different times. So we must decide what variables will be managed by which players at what time.

2.1.2 How Should Replicated Memory Be Updated?

One way a game developer can simplify communication is to replicate the state updating mechanism for each of the players. Some multi-player games use a protocol for negotiating the state when players "collide." However, if players only communicate their actions upon the environment, each player can use the same calculations to determine the resulting state of "collisions" to keep their states consistent. For example, if a player tries to grab a dungeon key, he can communicate "grab" and all of the interacting players can independently determine whether or not he picks up the key because they agree on the position of the key and player's position. In contrast, if two players communicate "grab key," there may be overhead to negotiate which player actually gets the key. So, by just communicating actions, we can eliminate the need to provide read and write access to shared variables. The general strategy we use, then, is to transmit only *state deltas* among players, each of which independently computes the states that result from the application of these deltas.

2.1.3 Hierarchies of Groups

Players outside region of interaction R do not need to receive the actions of the players within R, but they will need to know the state of variables geographically bound to R when they enter the region. The current values of the variables must be kept by players within R. It makes sense that players in the regions of the variables maintain their current state. By dividing players into groups corresponding to regions where state variables are available, consensus of local

state can be maintained through the total ordering of action messages. By fixing groups to regions of the environment, current variable states can be consistently maintained. Multiple non-overlapping groups can be responsible for partitions of the game state.

One problem with fixing groups to locations is that players can evacuate a region, taking the state of the region with them. Our approach is to form a hierarchy of groups so that when regions are evacuated, higher-level groups covering sets of smaller regions can save the state of the newly-empty region. When a player enters an empty region, players at the higher-level region can pass the state of the region that was current at the time it was evacuated. The alternative (having groups that move with the players) requires that non-interacting players communicate state updates. However, we must decide how the state is passed between groups on different levels. We do not want all players to be members of their higher-level groups since this would mean that all players receive state updates from the entire environment at the highest-level. Thus, a leader is designated as a member of the higher-level group to serve as a gateway between adjacent levels.

We could occasionally send state updates to the higher-level, so that when players enter the region, only the most recent actions need to be sent. However, it is more compact to send the values of the updated variables instead of sequences of players' actions. It is much simpler to just pass the full state when a player joins the group and only send updates when a region is evacuated.

2.1.4 How Big are Regions?

A simple way to divide up the environment into regions is to make them the size of the largest interaction area of the players. If we do this, then players can only interact with 2^d regions at a time for a d-dimensional environment. So, in a 2-dimensional environment, players can only be member of at most four lowest-level groups. We can stretch the size of the regions based on how fast players can move within the environment to lessen the number of group membership changes, but this is a tradeoff with the number of players that will likely be in same region. Another strategy is to buffer the player by making him join a certain number of surrounding regions to ensure that transitions into new regions are smooth. For a two-dimensional environment, we can ensure that a player is always a member of the four or nine surrounding regions. These parameters can be adjusted by the game developer to optimize the game's performance on top of Mr. Hide.

2.1.5 Implement with Dr. T?

One option that was considered for implementation was to spawn a Dr. T thread for each group that a player joins. Dr. T can provide causal-atomic ordering of messages needed to keep state variables consistent for individual regions. However, we cannot ensure that the code written by application-level developers is thread-safe, so this approach was abandoned. Instead, a group communication

service was developed to handle multiple group memberships and provide causal-atomic ordering of messages. In order to transfer state between groups, reliable point-to-point send and receive on top of UDP was developed.

2.1.6 Summary

The advantages of Mr. Hide can be summed up as follows: It provides a general communication service well-suited to a range of multi-player games and shared whiteboard applications. Clients of Mr. Hide can easily implement replicated shared memory with action messages, and communication is minimized for distributed applications with a spatial shared environment. The disadvantages of Mr. Hide are that state transfer overhead for membership changes might slow applications. Furthermore, Mr. Hide is not well suited to applications where all players access the same region simultaneously; the design of Mr. Hide assumes that this type of play is rare, and the optimizations of Mr. Hide will simply reduce performance in such cases. Finally, it was *very* difficult to implement Mr. Hide.

2.2 Mr. Hide Communications Protocol

The pseudocode below outlines the heart of the Mr. Hide event-handling system.

```
Deliver message.  Apply the following take appropriate action
based on message type:

JOIN message:
    If I am the leader of the group to be joined
        send ADD to group
        add player to group
        send group STATE and membership information to player
    Else if target group is in a region I lead that is empty
        send that groups STATE to player making him the leader
        join player to subgroup
    Else route the message to the player known closest
        to the target group

LEAVE message:
    If the leader is leaving the group
        If I am the leader and only member
            If I am the leader of the parent group
                Send group STATE to next leader of parent group
            Else
                Send group STATE to leader of the parent group
            send LEAVE message to parent group
        Else if I am the next leader of the group
            send JOIN message to parent group
```

```
    remove player from my membership information
    If I am the sender, leave group

ADD message:
    If not joining lowest level group
        add player to membership information as leader of
            lower-level group from which he's joining

STATE message:
    update state information for regions covered by the group
    update membership information for group and subgroups
    If joining empty subgroups
        add self as leader of subgroups to membership information

ACTION message:
    update state for received players action (move) in game
    check new position
        send JOIN to new regions entered
        send LEAVE to leader of regions leaving

INIT message:
    If I am highest-level group leader
        initialize player in game state
        send STATE information to player joining game
    Else route message up hierarchy
```

2.3 Mr. Hide Programming API

Mr. Hide's API has changed somewhat since first described in a preliminary design document [6]. The major addition is the requirement that a `serialize_region()` and an `unserialize_region()` function be passed in the parameters to `mrh_register()`. These functions are needed for passing partial state information between regions based on group membership changes. Below we list the Mr. Hide interface. It is similar in spirit to the interface of Dr. T: clients of the module provide function pointers to Mr. Hide. Mr. Hide manages an event handler which calls these functions in response to network and keyboard events. See [18] for a description of the meaning of the function pointers required by Dr. T. The Mr. Hide interface is designed to be very similar.

```
extern void
mrh_register(void (*initialize_process)(int),
             void (*before_update)(void),
             void (*update)(int, int),
             int  (*get_input)(void),
             void (*after_update)(void),
             void (*serialize_state)(void **, int *),
```

```
            void (*unserialize_state)(void *, int),
            void (*serialize_region)(int, int, int, int,
                                     void **, int *),
            void (*unserialize_region)(int, int, int, int,
                                       void *, int),
            int dim,   // dimension of game environment
            int environment_size[],
            int max_interaction_area[],
            int interaction_area_buffer[],
            void (*new_regions) (char *, char *, char *),
            void (*old_regions) (char *, char *, char *),
            void (*location) (char *),
            int cell_buffer);

// These help determine region indices from locations
void region_bounds(int rx, int ry, int &bx, int &ex,
                   int &by, int &ey);
void region(int x, int y, int &rx, int &ry);
```

Mr. Hide implements all of the serialize functions as defaults as part of a service for managing game state variables. A GameState object makes the following functions available for reading and writing to local memory and packaging data for state transfers so that the game developer can ignore details about environment divisions. Below we list the class header file for Mr. Hide's GameState class.

```
// Etype must have a constructor with no arguments,
// accessor functions for object locations,
// partial_pass and partial_pass_loc public attributes
// to distinguish between dynamics of object types,
// public stuff() and unstuff() member functions,
// to aid with serializing state, and a copy constructor.
//
// int partial_pass, int partial_pass_loc;
// void GameObjectType::get_x_coord();
// void GameObjectType::get_y_coord();
// void GameObjectType::stuff( char *);
// void GameObjectType::unstuff( char *);

template <class GameObjectType>
class GameState {
public:

    // Game objects are subdivided:
    // array of player objects
    Vector<GameObject> players(MAX_PLAYERS);
```

```
    // array of unchanging objects
    Vector<GameObjectType> no_pass(MAX_NO_PASS);
    // array fixed-location objects
    Vector<GameObject> no_loc(MAX_NO_LOC);
    // array of dynamic objects
    Vector<GameObject> pass_all(MAX_PASS_ALL);

    // indices to next available slot in object arrays
    int players_next;
    int no_pass_next;
    int no_loc_next;
    int pass_all_next;

    // Constructors
    GameState();
    GameState(int,int);
    ~GameState(){};

    // Memory managemant functions
    int Insert(GameObjectType &);
    int InsertPlayer(GameObjectType &, int);
    int Remove(GameObjectType &);
    List<GameObject> & Read(int x, int y);
    int WipeMemory();
    int WipeMemory(int,int); // in region
    int Serialize(char *);
    int Unserialize(char *);
    int SerializeRegion(char *, int, int);
    int UnserializeRegion(char *, int, int);
};
```

3 Group Communications

One part of our project was to design and implement group communication and membership services. We designed a separate, generic group communication layer (GComm). It provides basic services to Mr. Hide in the form of API calls (`join()`, `leave()`, `gsend()`, `deliver()`) and creates a process group abstraction to the Mr. Hide above. The GComm layer implements a closed version of groups where only members can send messages to the whole group. The membership protocol also implements a closed version of membership where only current member of a group can invite new members. Nobody from the outside can join the group without such invitation. The Mr. Hide layer above must manage when to call the `join()` function to invite new members to the group. In case of our project, the Mr. Hide layer performs that function by sending higher level point-to-point JOIN messages. The GComm layer also allows each

process to be a member of an arbitrary number of groups.

The overall protocol is similar to Totem's single ring protocol. The GComm layer uses a token to carry sequence numbers and a process can send messages only while holding a token. This section describes the GComm protocol at a very high level, the implementation of group multicast using broadcast facilities, the GComm API, problems we encountered during the implementation, and the current state of GComm.

3.1 The GComm Protocol

The GComm protocol is based on a token-ring. Each group has a token associated with it that circulates among all the group members. The token carries message sequencing information used for the safe delivery of messages (safe delivery has the same semantics as Totem's safe delivery). The token also carries membership information whenever a process joins or leaves the group. The high level steps in the protocol are:

- receive token
- update message sequencing information
- send all multicast messages waiting on the list
- check if any previously received messages can be safely delivered and if so, make them ready for delivery
- send the token to the next process on the ring

3.2 Multicast Implementation

GComm takes advantage of UDP broadcast as the base for its multicast service. All multicast messages are sent while a process is holding the token. The multicast is implemented in the following way:

- send a broadcast message on the LAN
- scan membership list (which is sorted by IP addresses) to see if there are any group members on different LANs
- for each new LAN, send a point-to-point message to one of the group members on that LAN
- when a process receives such point-to-point message, it broadcasts it in turn on it's own LAN

When a process receives a multicast message intended for a group of which it is not a member, the message is discarded. This design optimizes the use of available resources. Some design considerations will be presented at the end of this section.

3.3 GComm API

The GComm layer provides the following functions as its API:

- `join(group_name, new_ip)`
  `group_name` is the name of the group. The Mr. Hide layer above assigns group names and understands their semantics. The restriction imposed by GComm is that group names must be character arrays. The maximum length of group name is provided during initialization. `new_ip` is the IP address of the process that is about to be invited. This function adds the new member to the membership list and makes sure that all group members receive this updated information.

- `leave(group_name)`
  This function removes the process from the given group and passes that information to other members.

- `gsend(group_name, message, length)`
  This function gives the application process the ability to send group (multicast) messages. It takes the message and saves it on a list of messages to be broadcast when the token associated with the given group arrives.

- `deliver(group_name, message, length)`
  This function implements the safe delivery of messages. It returns a message in "safe order" if there are any messages that can be safely delivered. If there are no such messages, it returns a NULL message and integer -1 to signal that nothing has been delivered.

3.4 Comments

When designing the GComm protocol, we were strongly influenced by the Totem protocol. The main attributes were its simplicity and clarity as well as the fact that it does not have a single point of failure (like protocols with central sequencer). GComm in its current version implements all the above feature with some limitations. The main restriction comes from the fact that reliable token transmission has not been implemented yet, which effectively restricts all group members to the same LAN because of packet loss when going over routers.

One of the issues that interfered with design and implementation of the GComm layer was the fact that we could not use multi-threaded code. The Solaris curses library is not thread-safe, and our sample application uses curses. This forced us to discard our initial design which used threads and create a new one that was efficient running in a single thread with the actual application. One possible alternative might have been to protect all curses code with synchronization primitives, but we did not explore this option.

One important issue that we did not to handle due to time constraints is fault tolerance. In the current version, if a process crashes, the entire group communication for all groups of which it was a member will fail.

4 Results

4.1 Game Application

The game application that runs on top of the Dr. T and Mr. Hide communication layers is a 2-dimensional grid-based game which includes scrolling, collision detection, and various items in the game which the player can interact with to modify their internal state. Players can attack other players by "running into" them, or by shooting at them (although shooting is not fully implemented in the current version). User input consists of keystrokes. This game was originally a single-host, single-player application and thus allows us to assess the ease with which our communications infrastructures can be retro-fitted onto existing applications.

4.2 Integrating with Dr. T

Surprisingly, it was quite straightforward to convert an existing single-player game into a multi-player distributed game with the Dr. T package. Rather than the game calling functions in a communication API, Dr. T calls functions which the game programmer writes. This may seem backwards at first, but since the set of functions passed into Dr. T mapped almost directly onto the main loop of the game program most of the code could be left unchanged.

A couple of items caused more substantial rewrites of game code. More state information had to be kept with the individual player, since Dr. T calls state-update functions with the player number as input. This would probably have already been implemented in the original game if it allowed multiple players. The only other major revision needed was to implement functions to serialize and unserialize the game state. All of the pertinent information needed to reconstruct the game state had to be given to players joining the game. But this may not be much of a burden on the game programmer: if the player can save and reload a game in single-player mode then serialize and unserialize are already written. Dr. T is definitely a useful tool: it relieves the game programmer from coding the communication layer underlying distributed game, and allows him to concentrate on the actual game application. We found that this could be done in an obvious and intuitive manner, as the functions that need to be written and passed into Dr. T are probably already present in the game code.

Although this simple interface seems to be appropriate for a broad class of games, there are cases where this simplification makes it difficult to control game execution. For instance, projectiles could simply be modeled as players, which could take turns moving with the other players. But Dr. T only permits one move per player, and if we want a projectile to move twice as fast as a player, but not miss spaces on the screen, a modification would have to be made to prevent player movement from occuring on every reception of the token. In our case there are problems with this approach: since the system's biggest limitation is its ability to scale with the number of players, introducing projectiles and other objects in the world as players is probably not the best solution. Instead,

the approach that we take is to use the `game_after_update()` function. This function is called for every player, every time around the loop, regardless of whether or not he actually moved. It is normally used for screen refreshes. Inside this function we keep a static clock or counter to determine how long it had been since projectiles last moved. If, when the function is called, a sufficient interval has passed, we advance our projectiles.

One other potential problem for the game is that in order to maximize the token speed, only the current move is sent for each player. But this assumes that each process will determine the same outcome for that player's movement given the current state of the game. This would seem to rule out any completely random occurrences, such as a sudden strong attack, or a "magically" appearing vial of health. But this could also be implemented by passing around a random number generator seed with the game state (if the game is running on different platforms, we would need to implement a portable version of rand). It may be slightly less intuitive to implement some things in Dr. T (and a distributed system) but for many games Dr. T provides the right level of control and abstraction.

4.3 Integrating with Mr. Hide

Integrating an existing application with Mr. Hide is slightly more involved than with Dr. T. Since Mr. Hide attempts to improve performance based upon localized interaction areas within the game, Mr. Hide requires more information than the simpler Dr. T interface. Like Dr. T, Mr. Hide uses Serialize and Unserialize to transfer the entire game state to a new player. But, Mr. Hide also includes more specialized serialize and unserialize functions for players who have joined the game (and have been given the global game state initially) and are simply moving between low level groups. In this case Mr. Hide takes advantage of two attributes of the objects in the world to determine exactly what information needs to be sent: whether or not the objects are stationary, and whether the objects can change state. Thus serializations which occur as a result of group membership changes will transmit less information. Of course the onus is on the game programmer to define all of the game objects in terms of these two attributes. For instance, in our game players can change internal state and move about in the environment as well. Health and ammo cannot move, but can change state, depending upon whether or not it has been used, and walls neither move nor change state.

The game programmer must also know about the partitioning of the game state, and must be able to ship off game state pertaining to a specific region (for my game Mr. Hide actually divides up the game state based upon the size of the players' "view area"). The game programmer could also define these low level groups himself, which would be more specialized to take advantage of specific features of the environment. This wouldn't make much sense for my simple 2-dimensional game, but might make more sense in a 3-dimensional game like Quake, where interaction areas are more complicated to determine, and Mr. Hide's default partition would be incorrect. Mr. Hide and the appli-

cation level are more tightly coupled and must share more information than the interface with Dr. T. This results in a slightly different interface (perhaps a bit more complex), but in cases where many players are fairly well distributed over a large game-world should also result in better performance and increased scalability as compared to the Dr. T approach.

5 Lessons Learned

Below we summarize in sound-bite format some of the lessons we learned — sometimes too late — during our work on this project. (For more of the same sort of wisdom, see the chapter on "bumper-sticker computer science" in [3].)

Simplicity is good.

Read manual pages carefully. The only way to avoid fatally botched designs is to read the fine print in the man pages before beginning detailed system architecture design. Be especially watchful for caveats and gotchas pertaining to weird, non-local interactions among different system calls and library routines. (Examples: you can't use multi-threaded code if you used curses, and you can't call a sleep function if alarms are pending.)

Rapid prototyping is good. It is very useful to write small pieces of throw-away code to obtain performance data and to allow simple proof-of-concept demonstrations. Prototypes are the right place to make the inevitable mistakes associated with doing anything for the first time. ("Plan to throw the first one out. You will, anyhow.")

Fanatical static checking is good in principle, but was difficult for us because Sun does not provide lint libraries for the curses and sockets code we used on our Solaris system. This reduced the effectiveness of the otherwise-excellent Solaris `lint`.

Assertions are good. The judicious use of assertions results in code which quickly debugs itself during testing.

Purify is good. However, run-time testing is only as good as the test inputs, and constructing stressful test inputs is a tricky art. Furthermore, Purify is a noisy channel because the Solaris sockets libraries frequently read uninitialized memory. This probably occurs when highly optimized routines copy data from user space to kernel space and probably does not indicate bugs, but it results in spurious warnings from Purify.

Correctness is very difficult to achieve, even for simple designs. Rigorous testing is required to expose subtle and hard-to-reproduce errors ("Heisenbugs") in protocols.

C++ isn't worth the bother. Good compilers are not yet available for the latest version of the language, and static checkers for C++ were not available on our Solaris system. In general C++ caused more problems than it solved.

Writing software in groups is difficult. Clean interfaces with clear semantics are essential. Furthermore, we should have adopted the Microsoft practice of frequently re-synchronizing different modules of an application while they are developed in parallel [10]. We waited too long to merge our modules together,

and this caused many problems that might have been prevented had we plugged prototypes together earlier.

6 Acknowledgments

The authors thank Professor Farnam Jahanian for several useful discussions and valuable pointers into the literature. Mike Bailey shared useful Java network timing data with us. The staff of CAEN went above and beyond the call of duty by very quickly supplying us with a map of the campus network topology.

References

[1] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membershop protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[2] John Bentley. *Writing Efficient Programs*. Prentice Hall, 1982. ISBN 0-13-970244-X.

[3] John Bentley. *Programming Pearls*. Addison-Wesley, November 1985. ISBN 0201103311.

[4] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[5] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. *Operating Systems Review (ACM)*, 27(5):44–57, December 1993. An excellent critique of CATOCS, illustrating why in many cases it at once provides too much and too little to application-level software.

[6] Brad Clement, Terence Kelly, Andrew Nierman, and Lukasz Opyrchal. Dr. T and Mr. Hide: Communications infrastructures for distributed interactive applications. Available on the Web at `http://www-personal.engin.umich.edu/~tpkelly/ds/`, November 1997.

[7] Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, volume I. Prentice Hall, third edition, 1995. ISBN 0-13-216987-8.

[8] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: Design, Implementation, and Internals*, volume II. Prentice Hall, second edition, 1994. ISBN 0-13-125527-4.

[9] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: Client-Server Programming and Applications*, volume III. Prentice Hall,

second edition, 1996. This book comes in three flavors: WinSock, TLI, and Berkeley sockets. We're using the Berkeley sockets version. ISBN 0-13-260969-X.

[10] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets*. The Free Press, July 1995. ISBN 0028740483.

[11] Ian F. Darwin. *Checking C Programs with lint*. O'Reilly & Associates, 1988. One of a very few books currently available on lint. Neither large nor recent, but provides a decent introduction to the basics. Also noteworthy for reprinting the Ten Commandments for C Programmers in an appendix. ISBN 0-937175-30-7.

[12] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, April 1995. ISBN 0-201-51459-1.

[13] David Evans. *The LCLint User's Guide*. MIT Laboratory for Computer Science, 2.2 edition, August 1996. LCLint is a powerful lint that uses special comments to convey the programmer's intentions to the verifier. Read all about it on the Web at `http://larch-www.lcs.mit.edu:8001/larch/lclint/`.

[14] Simson Garfinkel, Daniel Weise, and Steven Strassmann. *The UNIX-Hater's Handbook*. IDG Books, 1994. A bitterly humorous look at the dark side of Unix, written by knowledgeable programmers. Worth the purchase price for the chapter subtitles alone, e.g. "power tools for power fools" and, for the C++ chapter, "the COBOL of the 90's." ISBN 1-56884-203-1.

[15] Laurent Gautier and Christophe Diot. Mimaze, a multiuser game on the internet. Technical Report RR–3248, Institut National de Recherche en Informatique et en Automatique (INRIA), September 1997. Available on the Web at `http://www.inria.fr/rodeo/MiMaze/ReportInt.html`.

[16] Audio-Video Transport Working Group. Rtp: A transport protocol for real-time applications. Internet RFC 1889, January 1996. Available on the Web at `http://www.cis.ohio-state.edu/htbin/rfc/rfc1889.html`.

[17] David R. Hanson. *C Interfaces and Implementations*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997. An excellent book on modular and reusable software design by a true Jedi master of C (and a former professor of mine). All of the source code described in the book is available on the Web at `http://www.cs.princeton.edu/software/cii/`. The book itself is produced with the aid of Ramsey's `noweb` literate programming tool [33, 32].

[18] Terence P. Kelly. Dr. t: A communications infrastructure for distributed interactive applications. Available on the Web at `http://www-personal.engin.umich.edu/~tpkelly/drt/`, December 1997.

[19] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988. See section 2.5, page 41, for the ugly truth about integer division and modulus involving negative operands. ISBN 0-13-110362-8.

[20] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford, California, 1992. The 1995 printing is thought to be error-free. ISBN 0-937073-80-6. Details available on the Web at `http://www-cs-faculty.Stanford.EDU/~knuth/lp.html`.

[21] Andrew Koenig. *C traps and Pitfalls*. Addison-Wesley, 1989. Includes a brief discussion of C portability issues. ISBN 0-201-17928-8.

[22] Leslie Lamport. *LaTeX User's Guide and Reference Manual*. Addison-Wesley, second edition, 1994. ISBN 0-201-52983-1.

[23] Stephen A. Maguire. *Writing Solid Code*. Microsoft Press, 1993. Once you stop laughing at the idea that Microsoft is going to show us how to write good code, you'll learn a lot from this book. The basic argument is that software writers write too many bugs, and it's too expensive to catch and fix these bugs during testing. Therefore bugs must be caught as early as possible, ideally through automatic means. Maguire presents a series of steps that programmers can take to stop bugs as early as possible. Includes an entire chapter on the design and use of assertion macros. ISBN 1-55615-551-4.

[24] Stephen A. Maguire. *Debugging The Development Process*. Microsoft Press, 1994. Whereas [23] is for individual programmers, this book is for small software team leaders. ISBN 1-55615-650-2.

[25] David L. Mills. Network time protocol. Internet RFC 1305, March 1992. Available on the Web at `http://www.cis.ohio-state.edu/htbin/rfc/rfc1305.html`.

[26] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[27] Sape Mullender, editor. *Distributed Systems*. ACM Press Frontier Series. Addison-Wesley, second edition, 1993. ISBN 0-201-62427-3.

[28] J. Oikarinen and D. Reed. Internet relay chat protocol. Internet RFC 1459, May 1993. Available on the Web at `http://www.cis.ohio-state.edu/htbin/rfc/rfc1459.html`.

[29] P. J. Plauger. *The Standard C Library*. Prentice Hall, 1992. Provides a thorough description *and an implementation* of the ANSI standard C library. ISBN 0-13-131509-9.

[30] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992. Chapter 7, pages 274–316, is devoted to random number generation. ISBN 0-521-43108-5. There's also a *Numerical Recipes* Web page at `http://cfata2.harvard.edu/nr/nrhome.html`, where the most important books in the series are available in PostScript and PDF form.

[31] Pure Software, 1309 S. Mary Ave., Sunnyvale CA 94087. *Purify User's Guide.* An excellent introduction to an excellent tool. My edition is labelled part number PFY300-XPX-UGD. Read all about Purify at `http://www.rational.com/`.

[32] Norman Ramsey. *`noweb(1)`, `noweave(1)`, and `notangle(1)` manual pages.* University of Virginia. The `noweb` family of tools is installed on the University of Michigan CAEN system in `/afs/engin.umich.edu/u/t/p/tpkelly/bin/`. The `noweb` tools are freely available on the Web at `http://www.cs.virginia.edu/~nr/noweb/`.

[33] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994. Describes Ramsey's `noweb(1)` family of simple literate programming tools.

[34] Jeffrey Rothschild. The internet and performance. Technical report, Mpath Interactive, Inc., 1996. A white paper discussing performance issues in multi-player Internet games. Available on the Web at `http://www.mpath.com/news/technical.html` .

[35] Richard M. Stallman and Roland McGrath. *GNU Make.* Free Software Foundation, 59 Temple Place, Suite 330, Boston MA 02111, 0.50 edition, March 1996. ISBN 1-882114-79-5.

[36] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, 1990. ISBN 0-13-949876-1.

[37] W. Richard Stevens. *Unix Network Programming.* Prentice Hall PTR, second edition, 1998. The updated and expanded version of [36], hot off the press at the time of this writing. ISBN 0-13-490012-X.

[38] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, April 1995. ISBN 0-201-54330-3.

[39] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, August 1997. ISBN 0-201-88954-4.

[40] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995. ISBN 0-13-219908-4.

[41] Peter van der Linden. *Expert C Programming: Deep C Secrets.* Prentice Hall, 1994. A highly readable and humorous look into the dark corners

of C by a member of Sun's compiler and OS kernel group. Highly recommended. ISBN 0-13-177429-8. More information is available on the Web at `http://www.sun.com/books/books/vanderLinden/vanderLinden.html`.

[42] Kevin Watkins. *Discrete Event Simulation in C.* McGraw-Hill International Series in Software Engineering. McGraw-Hill, 1993. Includes code on diskette, none of which was used in the project described in this paper. ISBN 0-07-707733-4.

[43] Gray Watson. *Debug Malloc Library*, 3.2.0 edition, January 1997. Debug malloc is poor man's Purify. It's a drop-in replacement for the standard C library malloc. Detects many of the same errors as Purify (e.g. leaks, some forms of corruption). Available free on the Web at `ftp://ftp.letters.com/src/dmalloc/`.